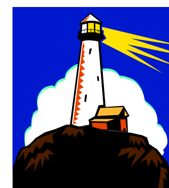
Network Programming: sockets

Antonio Lioy
< lioy@polito.it >

Politecnico di Torino
Dip. Automatica e Informatica

Warning for programmers

- **network programming is dangerously close to the O.S. kernel and consequently:**
  - can easily block the O.S.
    - diligence to control the results of all operations, without taking them for granted
  - the API can vary in minimal but important details
    - diligence to preview all cases to create “portable” programs
    - we’ll try to use Posix 1.g



Exercise - copy data

- **copy the content of the file F1 (the first argument in the command line) in the file F2 (the second argument in the command line)**

`copyfile.c`

Error messages

- **must contain at least:**
  - [PROG] name of the programe
  - [LEVEL] error level (info, warning, error, bug)
  - [TEXT] signalling of error most specifically possible (e.g. file name and input line number on which the error appeared)
  - [ERRNO] number and/or name of system error (if applicable)
- **suggested form:**

```
( PROG ) LEVEL - TEXT : ERRNO
```

Error functions

- **it is convenient to define standard error functions that takes as input :**
  - a format string for the error
  - a list of parameters to print out
- **UNP, appendix D.4**

	errno?	terminate?	log level
err_msg	no	no	LOG_INFO
err_quit	no	exit(1)	LOG_ERR
err_ret	yes	no	LOG_ERR
err_sys	yes	exit(1)	LOG_ERR
err_dump	yes	abort( )	LOG_ERR

errlib.h
errlib.c

stdarg.h

- **variable list of arguments (ANSI C)**
- **declared with variable-length argument list facility (. . .) as last argument of a function**
- **arguments used alltogether (ap) in appropriate functions (vprintf, vfprintf, vsprintf, vasprintf, vsnprintf) or one argument at a time (va_arg), but in this case it is necessary to find out in other way the number of arguments**

```
#include <stdarg.h>

void va_start (va_list ap, RIGHTMOST);
TYPE va_arg (va_list ap, TYPE);
void va_end (va_list ap);
```

From 32 to 64 bit: problems

- **evolvement of architectures from the ones on 32 bit to the ones on 64 bit modified also the dimension of data**
- **in particular it cannot be assumed anymore that | int | = | pointer |**
- **pay attention to use correctly the types defined to solve these problems (e.g. `size_t`)**

	ILP32	LP64
`char`	8	8
`short`	16	16
`int`	32	32
`long`	32	64
pointer	32	64

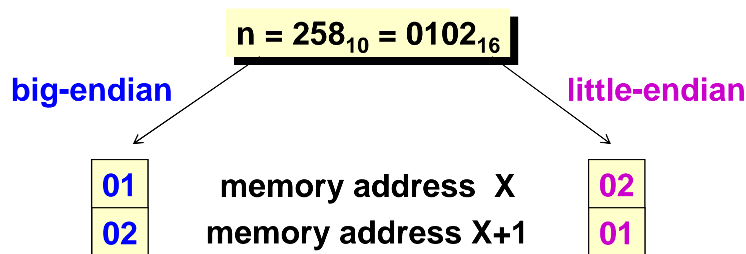
`datasize.c`

Data exchange among heterogeneous nodes

- **problem:**
  - when complex data are exchanged (that is not single ASCII characters) we cannot know with certainty that they are encoded in the same way on the various nodes
  - data encoding depends on HW + O.S.
- **solution:**
  - use a neutral format (network)
  - sometimes the conversion is done automatically by the functions …
  - … but often it is the explicit task of the programmer

Sources of data incompatibility

- different floating-point formats (IEEE-754 / non-IEEE)
- alignment of fields of a struct on a 1-, 2- or 4-byte boundary in memory (structure packing)
- byte order in integers (little-endian or big-endian)



byteorder.c

Functions "host to network"

- used for passing information to the network functions (e.g. ports and addresses) and that the network must understand
- not used for passing info of application data or when machines on both ends agree not to swap data and use the same byte order

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

uint32_t htonl (uint32_t hostlong);
uint16_t htons (uint16_t hostshort);
uint32_t ntohl (uint32_t netlong);
uint16_t ntohs (uint16_t netshort);
```

Note on integer types

- **sometimes are still encountered the old types:**
  - u_long (= uint32_t)
  - u_short (= uint16_t)
- **in the system cygwin are defined:**
  - u_int32_t (= uint32_t)
  - u_int16_t (= uint16_t)

- **advice:**
  - write programs using types uint32_t and uint16_t
  - if necessary, map them with a conditional #define

Network addresses

- **the IPv4 networks use directly addresses on 32 bit**
- **... not the names (e.g. www.polito.it), that are translated in addresses by the DNS**
- **... and not even addresses in dotted-decimal format (e.g. 130.192.11.51)**
- **... but its numerical value on 32 bit**
- **example: 130.192.11.51**
  - = $130*2^{24} + 192*2^{16} + 11*2^{8} + 51$
  - = (((130*256) + 192)*256 + 11)*256 + 51
  - = 130<<24 + 192<<16 + 11<<8 + 51
  - = 2,193,623,859

Conversion of network addresses

- generally, standard functions require the numerical address to be expressed in form of struct
- the functions inet_ntoa( ) and inet_aton( ) serve to convert numerical addresses from/to strings
- use of other functions (e.g. inet_addr) is deprecated

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

struct in_addr {
  in_addr_t s_addr
};
```

inet_aton ( )

- converts address in dotted-decimal format (“dotted notation”) …
- … from string
- … to numerical format (network)
- returns 0 if the address is not valid

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

int inet_aton (
  const char *strptr,
  struct in_addr *addrptr
);
```

inet_ntoa ( )

- converts address in dotted-decimal format ("dotted notation") …
- … from numerical format (network)
- … to string
- returns the address in string format

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

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

Example: validation of an IP address

Write a program that:

- takes as input on the command line an IP address in dotted notation
- returns its numerical value
- signals error in case of erroneous format or illegal address

Verification:

- remember that A.B.C.D = A<<24 + B<<16 + C<<8 + D

`avrfy.c`
`avrfy2.c`

Initialization of structures (ANSI)

- to treat data structures as sequences of bytes
- …that can include NULL and consequently cannot be manipulated with the "str…" functions (strcpy, strcmp, …)

```
#include <string.h>

void *memset (
  void *dest, int c, size_t nbyte )
void *memcpy (
  void *dest, const void *src, size_t nbyte )
int *memcmp (
  const void *ptr1, const void *ptr2, size_t nbyte )
```

Initialization of structures (BSD)

- functions pre-ANSI, defined in Unix BSD
- still widely used

```
#include <strings.h>

void bzero (
  void *dest, size_t nbyte )
void bcopy (
  const void *src, void *dest, size_t nbyte )
int bcmp (
  const void *ptr1, const void *ptr2, size_t nbyte )
```

Management of signals

- **signals are asynchronous events**
- **to each received signal corresponds an implicit default behaviour**
- **to change the response to a signal:**
  - modify the default behaviour
  - catch it by registering a signal handler

Timeout

- **sometimes it is necessary to start timeout counters to:**
  - wait in idle state for a pre-established period of time (sleep)
  - know when a certain period of time expired (alarm)

`sveglia.c`

sleep ( )

- activates a timer and suspends the process for the indicated period of time
- if it terminates because the pre-established period of time expired
  - returns zero
- if it terminates because interrupted by a signal
  - returns the time remaining up to the pre-established time

```
unsigned int sleep (
  unsigned int seconds
);
```

alarm ( )

- activate a timer and a SIGALRM signal is automatically generated when the timer expires
- the process is not suspended
- note: the default response to SIGALRM is to terminate the process that receives it
- ???a successive call cancels the current timer
- ???attention: unique timer for all processes of a group

```
void alarm (
  unsigned int seconds
);
```

The socket

- **it is the base function call (primitive) used in the TCP/IP communication**
- **it is the end (final) point of a communication**
- **suitable for channel-oriented communication:**
  - connected socket (a couple of connected sockets provides a bidirectional interface of type *pipe*)
- **suitable for message-oriented communication:**
  - non connected socket

Socket types

- **three fundamental types:**
  - STREAM socket
  - DATAGRAM socket
  - RAW socket
- **typically:**
  - stream and datagram are used at application level
  - raw used in the implementation of protocols (access to all fields of the IP packet, including the header)

Stream socket

- octet stream
- bidirectional
- reliable
- sequential flow
- non duplicated flow
- messagges of unlimited dimension
- interface of type sequential file
- usually used for TCP channels

Datagram socket

- message-oriented
- message = group of non structured bytes (binary blob – block of bytes)
- bidirectional
- non sequential
- not reliable
- duplicated (if necessary)
- messages limited to 8 KB
- specific interface (to message)
- usually used for UDP or IP packets

Raw socket

- **provides access to the lower communication protocol**
- **typically of type datagram**
- **complex programming interface (and often depends on the O.S.): not targeted for users of distributed applications**
- **used for the implementation of protocols**

Communication domain

- **a socket is defined in the frame of a communication domain**
- **a domain is an abstraction that implies:**
  - a structure for expressing addresses
    "Address Family" (AF)
  - a set of protocols that implement the sockets in the domain
    "Protocol Family" (PF)
- **commonly it is almost always used only AF (because it exists an unambigous/unique correspondence with PF)**

Binding

- a socket is created without an identifier
- no process can reference or access to a socket without an identifier
- before being able to use a socket it is necessary to associate it an identifier
  - = network address (for network socket)
  - = logical name (for O.S. socket)
- the binding operation establishes the address of the socket and makes it accessible in network
- the binding operation is binded to the protocol used
- explicit binding is usually done only by the server

Association (INET domain)

- two processes can communicate in network if among them there exists an association
- in the AF_INET domain an association is a structure of 5 components
- all the associations (structures above) must be unique

protocol (TCP, UDP, …)
IP address (local)
port (local)
IP address (remote)
port (remote)

Association (Unix domain)

- **two processes on the same Unix node can communicate among them a local association**
- **in the AF_UNIX domain an association is a structure of 3 components**
- **all the associations (structures above) must be unique**

protocol (channel, msg)
pathname (local)
pathname (remote)

Associations

- **are created with the system call bind( ) that creates half of an association**
- **the association is completed:**
  - by the server with accept( ), that removes the request from the queue and creates a socket for this connection; it is a blocking call
  - by the client with connect( ), that assigns also the local port; it is a blocking call

Connected socket (stream)

- **the creation of a connection is typically an asymmetric operation**
- **each process creates its own endpoint with socket( )**
- **the server:**
  - waits for incomming connections on its socket by calling listen( )
  - when a connection request arrives it accepts it with accept( ), which removes the request from the queue and creates a socket dedicated for the connection
- **the client connects to the server with connect( ), that carries out implicitly the binding assigning also the local port**

Stream socket: pre-connection

The client:
1. create a socket

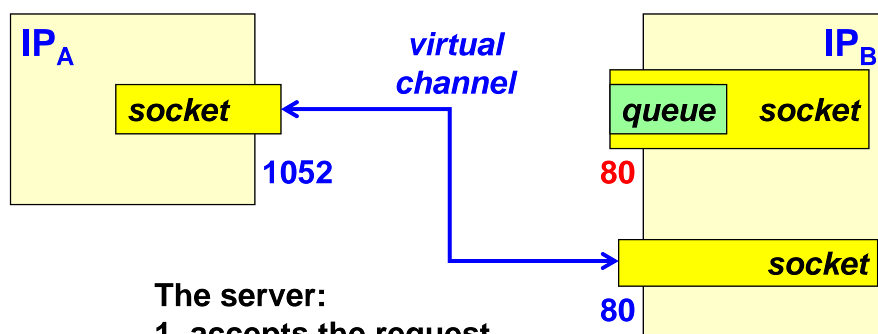
The server:
1. create a socket
2. associate a port to socket
3. listens for incoming request connections on the queue

Stream socket: connection request

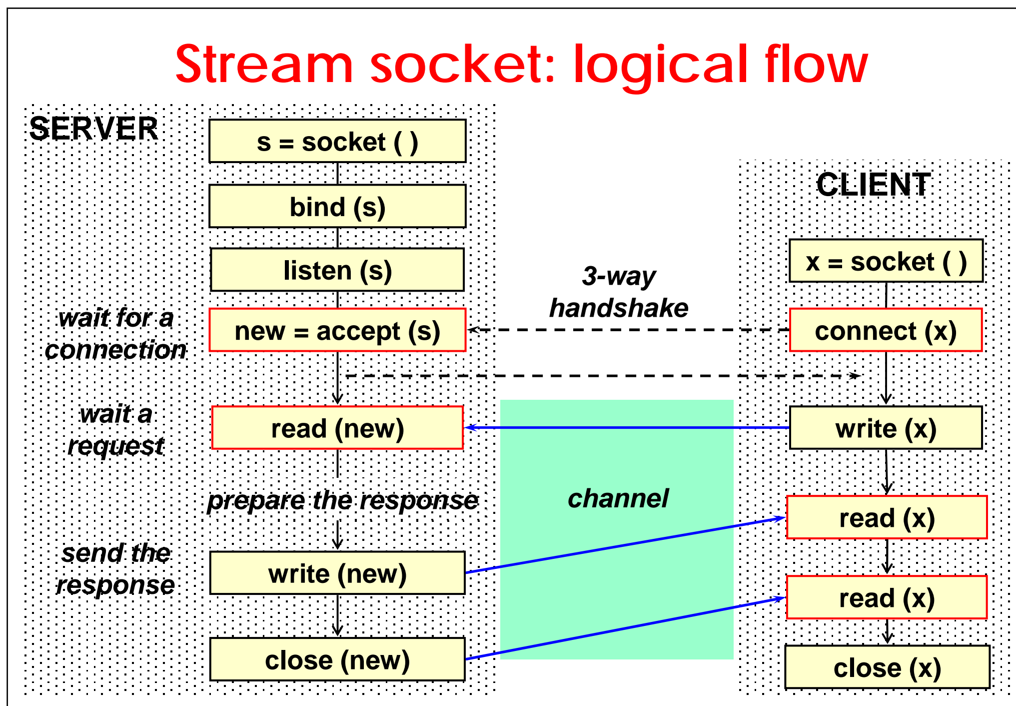
The client:
1. requests connection to the port of the server

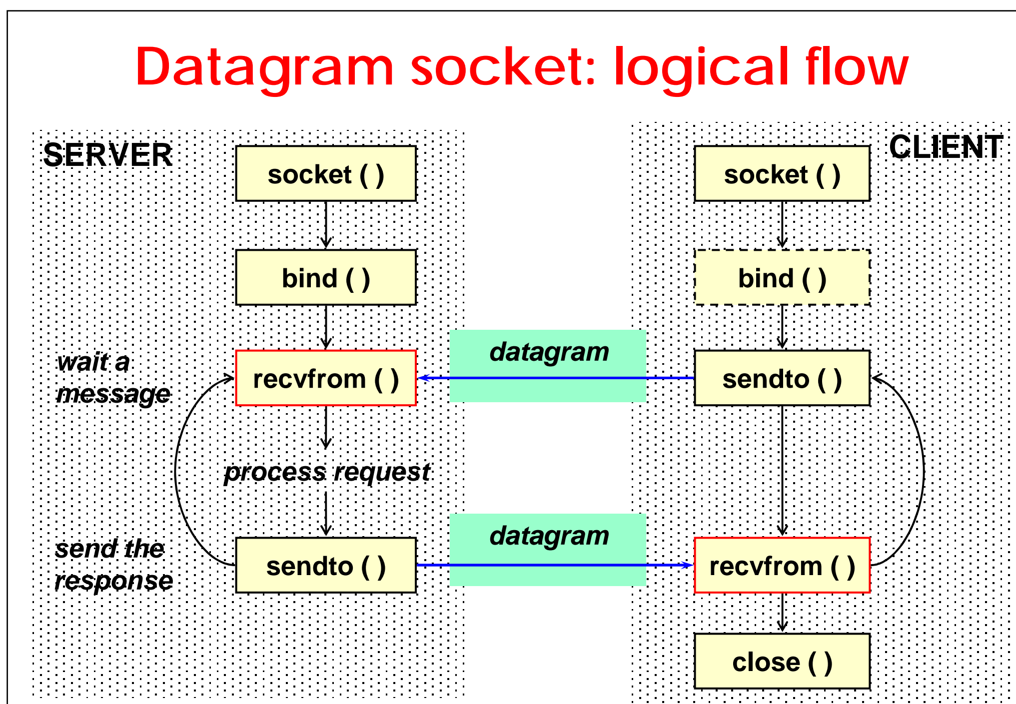
The server:
1. receives a connection request

Stream socket: established connection



The server:
1. accepts the request
2. completes the channel with a new socket
3. returns to listen for the incoming connections on the queue of the original socket


Stream socket: logical flow
SERVER
s = socket ( )
bind (s)
listen (s)
wait for a connection
new = accept (s)
wait a request
read (new)
prepare the response
send the response
write (new)
close (new)
3-way handshake
channel
CLIENT
x = socket ( )
connect (x)
write (x)
read (x)
read (x)
close (x)


Datagram socket: logical flow
SERVER
socket ( )
bind ( )
wait a message
recvfrom ( )
process request
send the response
sendto ( )
datagram
datagram
CLIENT
socket ( )
bind ( )
sendto ( )
recvfrom ( )
close ( )

Datagram socket - differences

- **allow to:**
  - exchange data without a connection (because the messages contain the destination address)
  - send from a socket to more than one destination
  - receive on a socket from more than one source
- **thus, in general, the model is "many to many"**
- **the terms client and server are used only in the sense of application**
- **there are no differences among the function calls made by the various processes involved in communication**

I socket in C

Note for compilation

- **in Solaris 8 programs must be linked also against the libsocket and libnsl libraries:**
  - gcc -lsocket -lnsl …

Unix socket descriptor

- **is a normal file descriptor referring to a socket instead of a file**
- **can be used normally for reading or writing**
- **it is possible to use the normal file system calls:**
  - close, read, write
  - exception: seek
- **other system calls available for specific socket functions (i.e. not applicable to file)**
  - send, recv, …
  - sendto, recvfrom, …

© A.Lioy - Politecnico di Torino (2003-4)

socket( )

- creates a socket
- returns the socket descriptor in case of success, -1 in case of error
- family = constant of type AF_x
- type = constant of type SOCK_x
- protocol = 0 (exception in raw socket)

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

socket( ): parameters

- **family:**
  - AF_INET
  - AF_INET6
  - AF_LOCAL (AF_UNIX)
  - AF_ROUTE
  - AF_KEY
- **type:**
  - SOCK_STREAM
  - SOCK_DGRAM
  - SOCK_RAW
  - SOCK_PACKET (Linux, access to layer 2)

Possible combinations

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	yes		
SOCK_DGRAM	UDP	UDP	yes		
SOCK_RAW	IPv4	IPv6		yes	yes

socket( ) : wrapper

- it is convenient to write only once the test for the return values instead of repeating them each time (and eventually forgetting them!)

```
int Socket (int family, int type, int protocol)
{
  int n;
  if ( (n = socket(family,type,protocol)) < 0)
    err_sys ("(%s) error - socket() failed", prog);
  return n;
}
```

The concept of "wrapper"

- **given a function to be "wrapped", create a function:**
  - namesake but starting with capital letter
  - with the same identical parameters
  - of type void, because the controls are made in the body of the new function
- **it is not necessary to invent "exotic" controls but only to read carefully the function definition and check the return value (or verify other mechanisms for error signalling)**

Example of wrapper

- **instead of strcpy( ) prefer always strncpy( )**
- **… but this function can also generate errors:**

```
char *strncpy (char *DST, const char *SRC, size_t LENGTH);

DESCRIPTION
'strncpy'  copies  not  more than LENGTH characters from the
string pointed to by SRC (including the terminating null
character) to the array  pointed  to  by DST.

RETURNS
This function returns the initial value of DST.
```

Wrapper for strncpy

```
void Strncpy (
  char *DST, const char *SRC, size_t LENGTH)
{
  char *ret = strncpy(DST,SRC,LENGTH);
  if (ret != DST)
    err_quit(
      "(%s) library bug - strncpy() failed",prog);
}
```

Address of a socket

- is used the structure sockaddr that represents the address of a generic socket (Internet, Unix, …)
- in practice is only an overlay for the specific cases (sockaddr_in, sockaddr_un, …)
- defined in <sys/socket.h>

```
struct sockaddr {
  uint8_t      sa_len; // not mandatory
  sa_family_t  *sa_family, // AF_xxx
  char         sa_data[14] // identifier
}
```

Address of an Internet socket

- network address (level 3)
- port (level 4)
- L4 protocol defined implicitly on the basis of the socket type (STREAM, DGRAM)

```
struct sockaddr_in
{
  uint8_t         sin_len; // not mandatory
  sa_family_t     *sin_family; // AF_INET
  in_port_t       sin_port; // TCP or UDP port
  struct in_addr  sin_addr; // IP address
  char            sin_zero[8] // not used
}
```

connect( )

- creates a connection among a “local” socket and a “remote” socket, specified by its identifier (=address and port)
- in practice starts the TCP 3-way handshake
- the operating system assigns automatically an appropriate identifier to the local socket (address and port)
- returns 0 if OK, -1 in case of error

```
#include <sys/socket.h>
int connect (int sockfd,
  const struct sockaddr *srvaddr, socklen_t addrlen)
```

bind( )

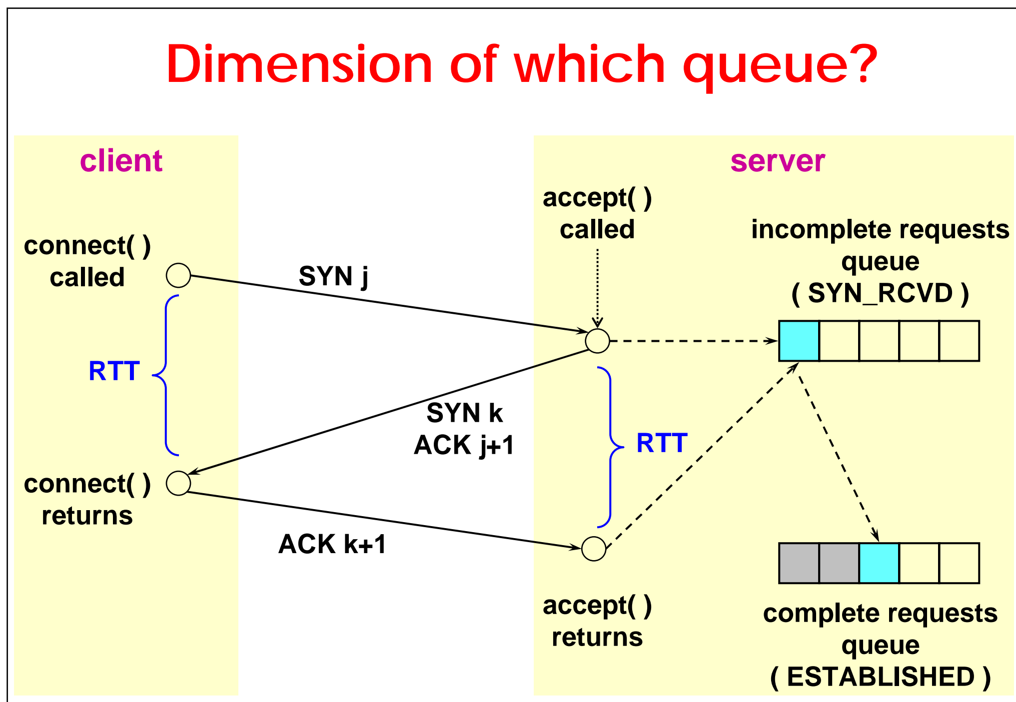
- assigns an address to a socket
- returns 0 in case of di success, -1 in case of error
- if the IP address is not specified, the kernel assigns it on the basis of the packet SYN received
- to specify any address it is used INADDR_ANY

```
#include <sys/socket.h>
int bind ( int sockfd,
  const struct sockaddr *myaddr,
  socklen_t myaddrlen )
```

listen( )

- transforms a socket from active to passive
- specifies the dimension of the queue containing pending incoming connection requests (= sum of the two queues, plus sometimes an adjusting factor 1.5)
- critical factor for:
  - the server with high load
  - resist to “SYN flooding” attacks
- returns 0 in case of success, -1 otherwise

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



Note on parameter "backlog"

- **in Linux starting with kernel 2.2 the parameter specifies only the length of the queue Established**
- **the length of the queue SYN_RCVD is:**
  - automatically determined (256-1024 entry) depending on the available RAM
  - changeable through sysconf or …/ip/…
  - infinite if the SYN_COOKIE are activated
- **in Solaris …**

listen( ) : wrapper

- it is convenient not to fix the dimension of the queue in the source code but make it changeable (via arguments or environment variables)
- by example with a wrapper:

```
#include <stdlib.h> // getenv()

void Listen (int sockfd, int backlog)
{
  char *ptr;
  if ( (ptr = getenv("LISTENQ")) != NULL)
    backlog = atoi(ptr);
  if ( listen(sockfd,backlog) < 0 )
    err_sys ("(%s) error - listen failed", prog);
}
```

accept( )

- takes the first connection available in the queue of pending requests
- blocks if there are no pending requests (exception: if the socket is non blocking)
- returns a new socket descriptor, connected with that of the client
- side effect: returns the identifier of the client that has connected (unless NULL is provided)

```
#include <sys/socket.h>
int accept (int listen_sockfd,
  struct sockaddr *cliaddr, socklen_t *addrlenp)
```

close( )

- **closes immediately the socket**
- **the socket is not accessible anymore to the process but the kernel will try to send remaining data (if any) and then executes the closing of the TCP channel**
- **behaviour changeable with SO_LINGER**
- **returns 0 if terminates with success, -1 in case of error**

```
#include <unistd.h>
int close ( int sockfd )
```

Stream communication

- **use non buffered I/O functions to avoid:**
  - to remain indefinitely in wait (input)
  - to end up communication earlier in case of NULL (output)
- **never use the standard I/O library**
- **in practice use the read( ) and write( ) system calls:**
  - operate on file descriptor
  - return the number of bytes read or written, -1 in case of error

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t nbyte )
ssize_t write(int fd, const void *buf, size_t nbyte)
```

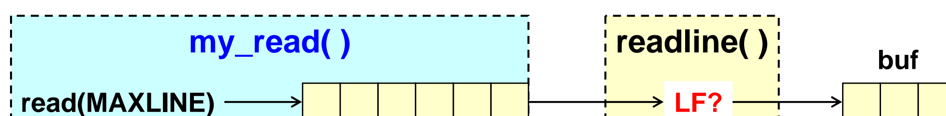
Result of a read on a socket

- > 0: number of bytes received
- = 0 : closed socket (EOF)
- < 0 : error

- attention:
  - because of the fragmentation and of buffers …
  - … the number of data bytes read can be less than the number of bytes expected to be received
  - it is thus convenient to write functions that process automatically this problem

"Secure" read / write functions

- **readn( )** and **writen( )** read and write exactly N bytes, unless an error or an EOF is encountered
- **readline( )** reads until an LF is encountered or untli the buffer is full, unless an error or an EOF is encountered
- reference: UNP, figg. 3.14, 3.15, 3.16
- note: readline( ) must necessarily read one byte at a time but uses another function (private) to fill in a buffer more efficiently



readn, writen, readline

- **the functions starting with capital letter control automatically the errors**

```
ssize_t readn (int fd, void *buf, size_t nbyte)
ssize_t Readn (int fd, void *buf, size_t nbyte)

ssize_t writen(int fd, const void *buf, size_t nbyte)
void Writen (int fd, const void *buf, size_t nbyte)

ssize_t readline (int fd, void *buf, size_t nbyte)
ssize_t Readline (int fd, void *buf, size_t nbyte)
```

sockwrap.h
sockwrap.c

Example: TCP daytime client and server

- **the service daytime (tcp/13):**
  - provides the current date and hour in a format that can be understood by a person
  - the responses are ended with CR+LF
- **implement a client that connects to the daytime service of the server specified in the command line**
- **implement an (iterative) server that receives service requests and returns the date and hour, identifying also the client that has connected**

daytimetcpc.c
daytimetcps.c

Attention!

- **because my_read uses a local buffer, the functions readline are not reentrant (that is it cannot be used in multiprocess or multithread environments)**
- **it is necessary to implement reentrant functions allocating externally the buffer for my_read**

Exercises

- **remove the function listen( ) from the server:**
  - what happens? why?
- **leave in place the function listen( ) but remove the function bind( ):**
  - what happens? why?

Information on socket

- **to find out address and port**
  - of the local side: getsockname( )
  - of the remote side: getpeername( )
- **return 0 if OK, -1 in case of error**

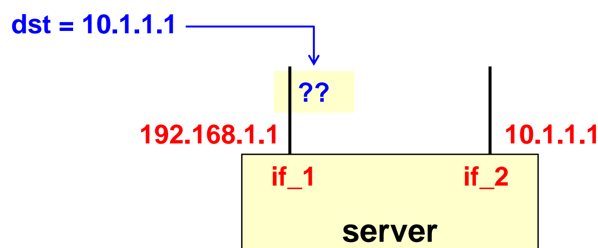
```
#include <sys/socket.h>
int getsockname ( int sockfd,
  struct sockaddr *localaddr, socklen_t *addrp )
int getpeername ( int sockfd,
  struct sockaddr *peeraddr, socklen_t *addrp )
```

Implicit or explicit binding?

- **consider the following case:**
  - server web multihomed (N network addresses)
  - a different web site for each address
- **if the server makes binding to INADDR_ANY:**
  - only one process is in listen( )
  - demultiplexing is made by the server
- **if the server makes binding to single N addresses:**
  - N processes in listen( )
  - demultiplexing made by the stack TCP/IP

weak-end / strong-end model

- in case of server multihomed …
- "strong-end model" = refer to the kernel that accepts packets on an interface only if DST_IP is equal to the IP of the interface
- "weak-end model" = refer to the kernel that accepts packets only if DST_IP is equal to the IP of any interface of the server



Ports dedicated to server and client

- IANA
  - 1-1023 = well-known ports
  - 1024-49151 = registered ports
  - 49152-65535 = dynamic / ephemeral ports
- UNIX
  - 1-1023 = reserved only to processes with EUID=0
  - 513-1023 = reserved to privileged clients (r-cmds)
  - 1024-5000 = BSD ephemeral ports (few!)
  - 5001-65535 = BSD servers (non-privileged)
  - 32768-65535 = Solaris ephemeral ports
- Windows instead does not make any control

recvfrom( ) and sendto( )

- used for the datagram sockets, even if they can be used also with stream sockets
- return the number of bytes read or written, -1 in case of errore
- the parameter "flags" is usually zero (further details later on)

```
#include <sys/socket.h>
int recvfrom ( int sockfd,
  void *buf, size_t nbytes, int flags,
  struct sockaddr *from, socklen_t *addrlenp )
int sendto ( int sockfd,
  const void *buf, size_t nbytes, int flags,
  const struct sockaddr *to, socklen_t addrlen )
```

Receiving data with recvfrom( )

- receiving zero data is OK (=payload UDP empty) and does not signal EOF (does not exist in socket datagram!)
- using NULL as value for the field "from" it is accepted data from anybody ... but then it is not known the address to respond to (!), unless this is received at the application level

Binding with datagram socket

- usually the client does not execute bind( ) but to the socket is assigned automatically an address and a port the first time it is used
- alternatively the client can execute bind( ) to the port 0 to indicate to the kernel to assign it a random temporary port

Example: UDP daytime client and server

- the service daytime (udp/13) provides the current date and hour in a format that can be understood by a person
- the server sends date and hour in an UDP packet to any client that sends to the server any UDP packet, even an empty one (with no data)
- develop a client that connects to the service daytime of the server specified in the command line
- develop an (iterative) server that receives service requests from clients and provides date and hour, indentifying also the client that has connected

`daytimeudpc.c`
`daytimeudps.c`

cygwin: known problems

- **sendto( ) with buffer of dimension zero does not send anything (should send an UDP packet with payload of length zero)**

Problems of datagram socket

- **problem 1: because UDP is not reliable, the client risks to remain blocked for ever in receiving state**
- **using a timeout solves the problem only in some cases:**
  - OK if sending again the request does not create problems
  - unacceptable otherwise (e.g. transaction of debit or credit)
- **problem 2: how to verify that the response comes exactly from the server to which we sent the request?**
- **the responses must be filtered at user level or at kernel level**

Verification of datagram responses

- **binary comparison among the address of the responder with the one of the original destination**
- **possible problems with multihomed server**
- **solution 1: make the verification not on the address but on the DNS name**
- **solution 2: the server makes an explicit binding to all its addresses and then listens on all of them (select)**

```
n = recvfrom(sfd,*buf,nbyte,0,(SA*)&from,&fromlen);
if ( (fromlen == serverlen) &&
     (memcmp(&servaddr,&from,serverlen) == 0) )
  // accepted response
```

Asynchronous errors

- **when an error appears in an UDP transmission (e.g. port unreachable) an ICMP error packet is generated …**
- **… but the function sendto( ) has already terminated with status OK**
- **… and consequently the kernel doesn't know to which application it should provide the error message (and in which way!)**
- **possible solutions:**
  - use connected datagram sockets (!)
  - catch the ICMP errors with a proprietary demon

Connected datagram sockets

- **it is possible to call connect( ) on a datagram socket to specify once for all the peer intended for further communication**
- **consequences:**
  - not used anymore sendto( ) but write( ) or send( )
  - not used anymore recvfrom( ) but read( ) or recv( )
  - the asynchronous errors that appear are returned to the process that controls the socket
  - the kernel carries out automatically the filtering of responses by accepting only packets from the peer
- **in case of repeated communication with the same peer, a performance improvement is noticed**

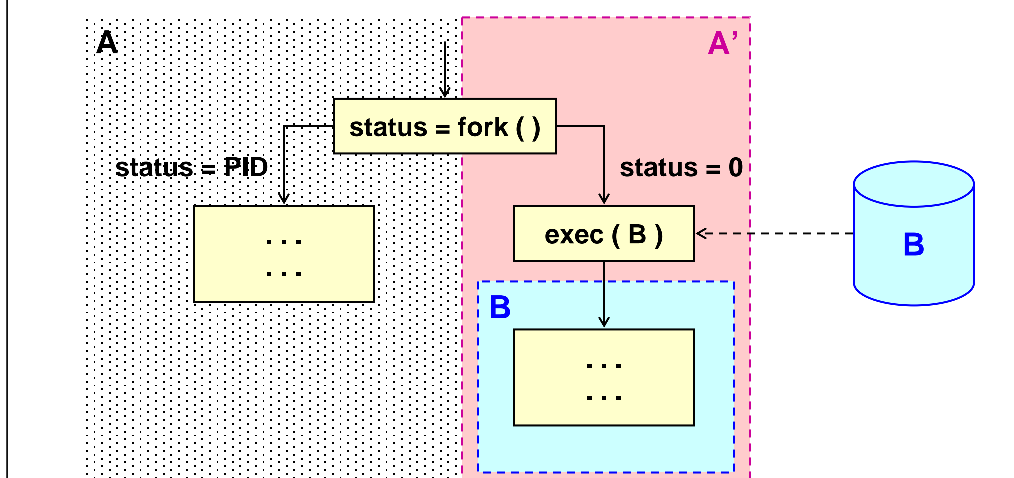
Disconnecting a datagram socket

- **can be done a second connect( ) towards another address to change the connection**
- **forbidden for stream socket**
- **to disconnect completely (that is turn to a non connected socket) execute connect( ) towards an address not specified by putting AF_UNSPEC**
- **in this case an EAFNOSUPPORT error could be received but it can be neglected**

Concurrent servers

Processes in Unix

- splitting of an application: fork( )
- possible start of a new image: exec( )



fork( )

- **generates a new process …**
- **… that shares image (source code) and execution environment with the process that executed fork( )**
- **return status:**
  - -1 in case of error
  - 0 for the new process (“child”)
  - PID of the new process to “father” (or “parent”)

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

getpid( ), getppid( )

- **the child process can find out the PID of the parent process with getppid( )**
- **note: the parent process of each process is unique**
- **to find out the own PID use getpid( )**
- **return status:**
  - -1 in case of error
  - the PID

```
#include <unistd.h>
pid_t getpid (void);
pid_t getppid (void);
```

The exec functions

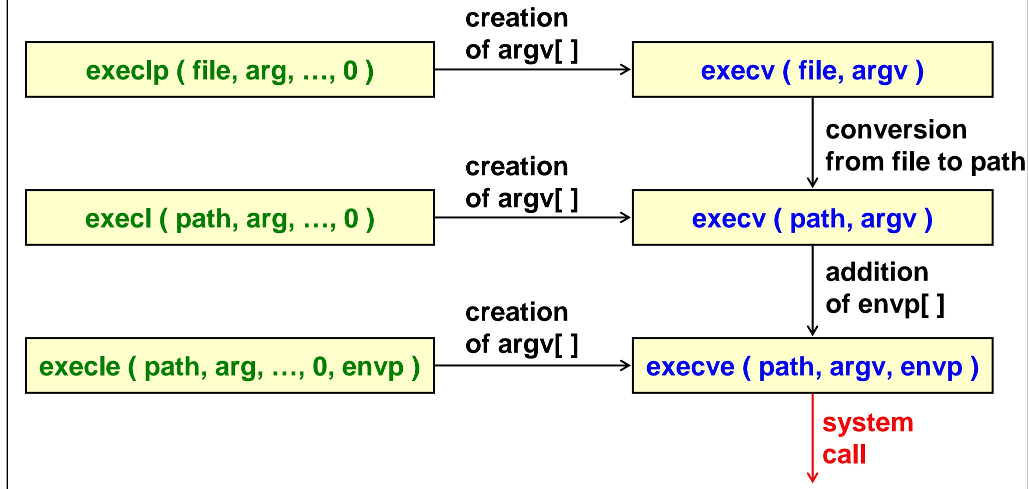
- replace the image in execution with a new image
- return -1 in case of error
- the functions L pass in the arguments as list of variables, terminated with NULL
- the functions V pass in the arguments as array of pointers, with the last element NULL
- the functions P localize the image throughout PATH, the others require the complete pathname
- the functions with E receive the environment variables as array of pointers, with the last element NULL; the others use the external variable "environ"

Functions execv( ) and execl( )

```
#include <unistd.h>
int execv ( const char *filename,
  char *const argv[] );
int execve ( const char *filename,
  char *const argv[], char *const envp[] );
int execvp ( const char *pathname,
  char *const argv[], char *const envp[] );
int execl ( const char *filename,
  const char *arg0, ..., (char *)NULL );
int execle ( const char *filename,
  const char *arg0, ..., (char *)NULL,
  char *const envp[] );
int execlp ( const char *pathname,
  const char *arg0, ..., (char *)NULL,
  char *const envp[] );
```

Functions exec

- **usually only execve( ) is a system call**



Communication by means of exec( )

- **explicit transfer of parameters:**
  - by means of the arguments
  - by means of the environment variables
- **the file descriptors (file and socket) remain open, unless the caller uses the function fcntl( ) to set the flag FD_CLOEXEC that determines to close them automatically when an exec is run**

Skeleton of concurrent server (I)

```
pid_t pid; // PID of child
int listenfd; // socket for listen
int connfd; // socket for communication

// creation of the socket for listen
listenfd = Socket( ... );
servaddr = ...
Bind (listenfd, (SA*)&servaddr, sizeof(servaddr));
Listen (listenfd, LISTENQ);
```

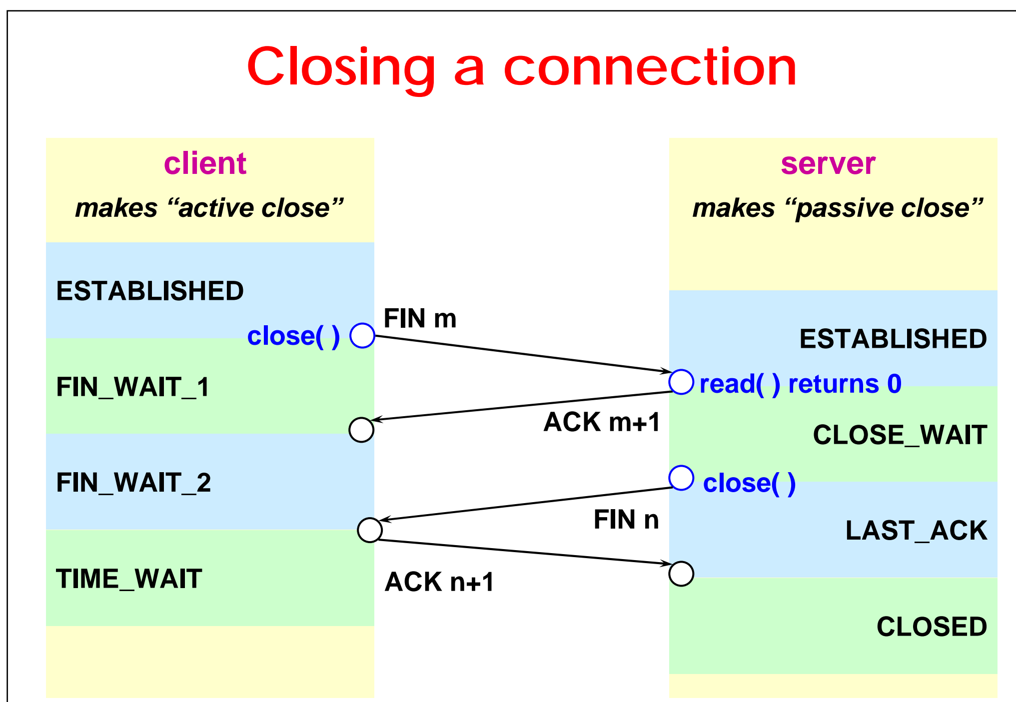
Skeleton of concurrent server (II)

```
// loop of execution on server
while (1)
{
  connfd = Accept (listenfd, ...);
  if ( (pid = Fork()) == 0 )
  {
    Close(listenfd);
    doit(connfd); // the child carries out the work
    Close(connfd);
    exit(0);
  }
  Close (connfd);
}
```

Importance of close( )

- **if the parent forgets to close the connection socket …**
  - exhaust the descriptors in a short period of time
  - the channel with the child remains open even when the child terminated and closed the connection socket
- **note:**
  - the function close( ) does not close the socket but it simply decreases the reference count REFCNT
  - only when REFCNT becomes zero, the socket is closed (that is FIN is sent)

Closing a connection



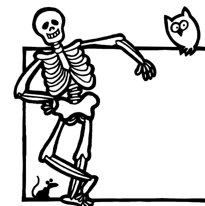
The TIME_WAIT status

- **can go out from the TIME_WAIT status only for timeout:**
  - time period equal to 2 x MSL (Max Segment Lifetime)
  - MSL = 2 minutes (RFC-1122), 30 seconds (BSD)
  - consequently timeout 1…4 minutes
- **exist to solve two problems:**
  - implement the closing of a TCP connection full-duplex
    - the last ACK could be lost and the client receives a new FIN
  - allow duplicated packets to “expire”
    - could be interpreted as part of a new form of the same connection

Termination of children processes

- **when a child process exits, a SIGCHLD signal is sent to the parent process**
- **parent’s default behaviour:**
  - ignored
  - … this determines the creation of a “zombie” process
- **zombies are inherited and eliminated by the init process only when the parent process exits (but usually the server processes never exit …)**

If we want to avoid zombies,
we have to wait for our children.
-- W.R.Stevens



Management of signals

- **pay attention at the differences of semantics in the various O.S.**
- **use per signum (by sign of) the logical names of signals (SIGCHLD, SIGSTOP, …)**
- **handler can be:**
  - a function defined by the user
  - SIG_IGN (ignore the signal)
  - SIG_DFL (default behaviour)

```
#include <signal.h>
typedef void Sigfunc(int);
Sigfunc *signal (int signum, Sigfunc *handler);
```

Remarks on signals

- **SIGKILL and SIGSTOP cannot be either caught or ignored**
- **SIGCHLD and SIGURG are ignored by default**
- **in Unix the signals can be also generated manually with the command**

  `kill -`***signal pid***
- **for example to send SIGHUP to the process 1234:**

```
kill -HUP 1234
```

wait( ) and waitpid( )

- **return:**
  - the PID of the child process that terminated; 0 or -1 in case of error
  - the exit status of the child process
- **wait( ) is blocking**
- **waitpid( ):**
  - does not block id the option WNOHANG is used
  - allows to specify the PID of a specific child process (-1 to wait for the first child that terminates)

```
#include <sys/wait.h>
pid_t wait (int *status);
pid_t waitpid (pid_t pid, int *status, int options);
```

Catching SIGCHLD

- **if several child processes terminate “simultaneously” only one SIGCHLD is generated, consequently parent must wait for all of them**

```
#include <sys/wait.h>
void sigchld_h (int signum)
{
  pid_t pid;
  int status;
  while ( (pid = waitpid(-1,&status,WNOHANG)) > 0)
#ifdef TRACE
    printf ("child %d terminated with status %d\n",
      pid, status)
#endif
  ; // pay attention at this semicolon
}
```

interrupted system call

- **when a process executes a “slow”system call (that is one that can block the caller)**
- **… can unblock not because the system call has terminated**
- **… but because a signal was caught; this case is signalled by errno == EINTR (or ECHILD)**
- **it is required to make provision for this case and repeat the system call**
- **very important to be considered for the accept( ) in server processes**
- **ATTENTION: all system calls can be repeated except for connect( ); in this case the function select( ) must be used**

Concurrent server: example

- **develop a concurrent server listening on the port tcp/9999 that receives text lines containing two integer numbers and returns the sum**
- **develop a client that:**
  - reads text lines from the standard input
  - send them to the port tcp/9999 of the server specified in the command line
  - receives response lines and print them on standard output

`addtcps.c`
`addtcpc.c`

Robustness of applications

Slow server

- **if a server is slow or overloaded …**
- **… the client can complete the 3-way handshake and then end up the connection (RST)**
- **… in the time period the server employs between listen and accept**
- **this problem**
  - can be processed directly by the kernel or can generate EPROTO / ECONNABORTED in accept
  - frequent case encountered in the web servers that are overloaded

Termination of server-child

- **when the server-child that communicates with the client terminates properly (exit) the socket is closed and consequently:**
  - a FIN is generated, accepted by the client's kernel but not transmitted to the application until the next
  - if the client executes a write it will receive a RST
  - depending on timing, the client will receive either an error on write or an EOF or ECONNRESET on the next read

SIGPIPE

- **process that executes write on a socket that received RST, receives the SIGPIPE signal**
  - default: terminate the process
  - if it is caught or ignored, the next write generates the error EPIPE
- **attention:**
  - if there is more than one socket open in write …
  - … SIGPIPE does not signal which one generated the error
  - thus it is better to ignore the signal and receive EPIPE on write

Crash of the server

- **covers also the case of unreachable server**
- **the write operations of the client work (there is no one to respond with an error!)**
- **the client's read operations go in timeout (sometimes after a lot of time: in BSD 9 m!) and generate ETIMEDOUT**
- **read and write could receive EHOSTUNREACH or ENETUNREACH if an intermediary router notices the problem and signals it via ICMP**
- **solution: set a timeout**
  - directly on the socket with the options
  - by means of select( )
  - by means of alarm( )

Crash and reboot of the server

- **sequence:**
  - crash (= unreachable server)
  - boot (= reachable server raggiungibile but "lost" existent connections: RST)
- **as a consequence read and write fail with ECONNRESET**

© A.Lioy - Politecnico di Torino (2003-4)

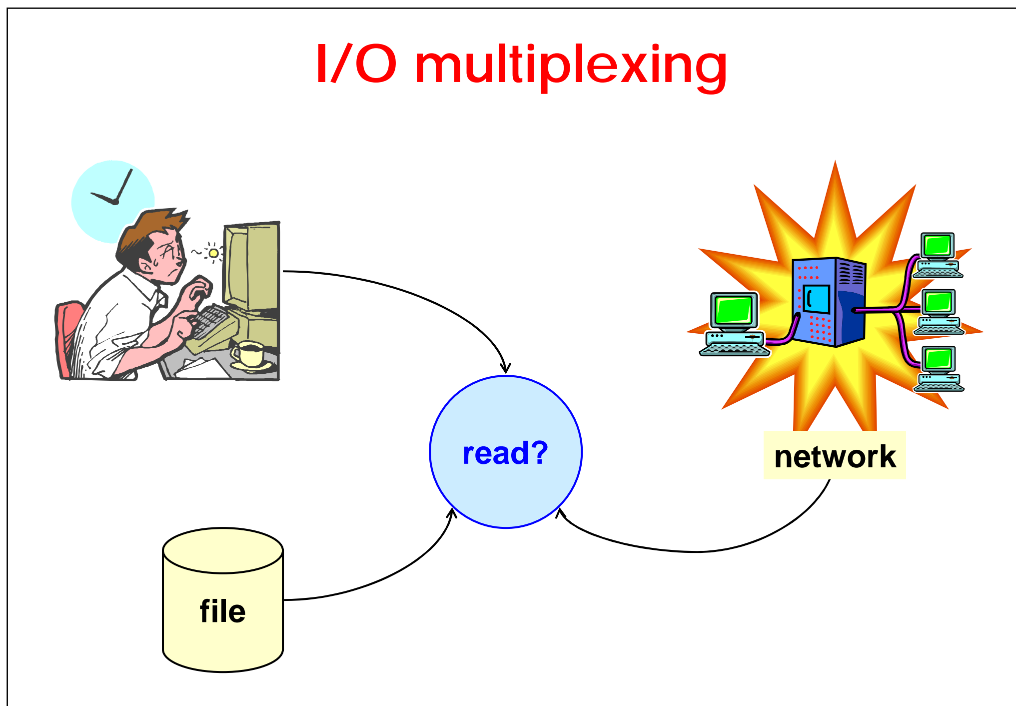
Shutdown of the server

- **at shutdown of a Unix node, the process init:**
  - sends SIGTERM to all active processes
  - after 5…20 seconds sends SIGKILL
- **SIGTERM can be caught and consequently the servers can attempt to close all sockets**
- **SIGKILL cannot be caught, terminates all processes and closes all open sockets**

Heartbeating

- **if it is necessary to find out as soon as possible if the peer is unreachable or broken it must be activated a "heartbeating" mechanism**
- **two possible implementations:**
  - by means of the option SO_KEEPALIVE
  - by means of heartbeating application protocol

I/O multiplexing



Applications of I/O multiplexing

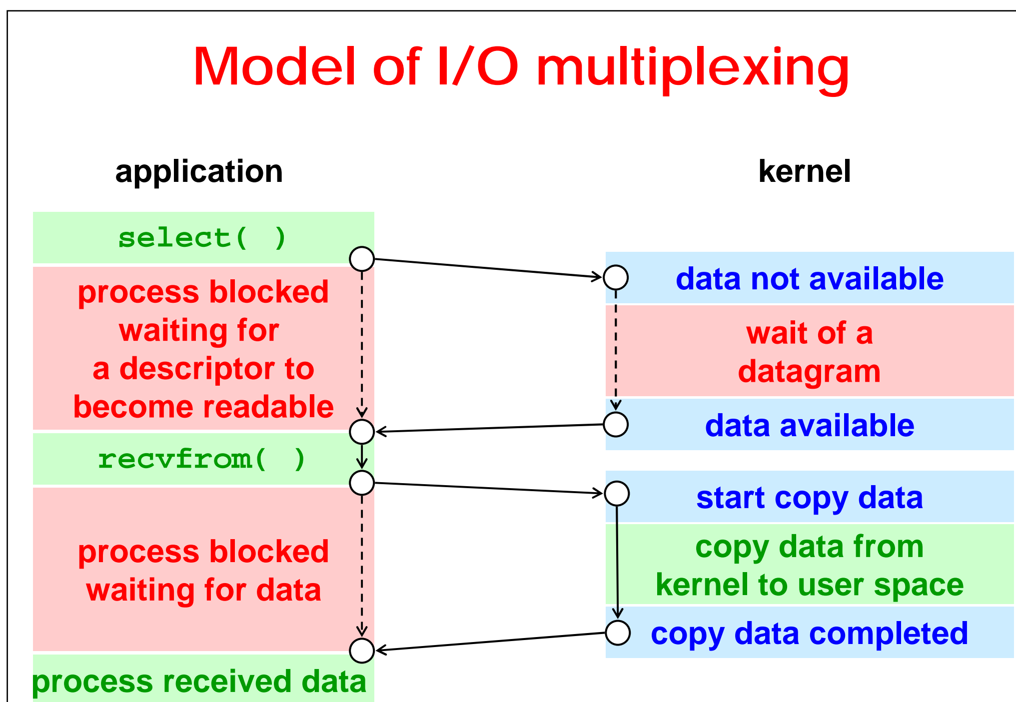
- a client that manages input from more than one source (typically user from the keyboard and server from the socket)
- a client that manages more than one socket (rarely, but typical in the web browsers)
- a TCP server that manages both the listening socket and the connection sockets (without activating separate processes)
- a server that manages both UDP and TCP
- a server that manages many services and/or protocols (rarely, but typical for the process inetd)

I/O Models

- blocking (read on normal sockets)
- non blocking (read on non blocking sockets)
- multiplexing (select, poll)
- signal-driven (SIGIO)
- asynchronous (functions aio_xxx)

- all pass through two phases:
  - wait for the data to be ready
  - copy the data from the kernel space to user space
- the problem is almost always in the reading phase, rarely in writing

Model of I/O multiplexing



select( )

- **blocks the calling process until …**
  - one of the selected descriptors becomes “active”
  - or the timeout expires (if set)
- **maxfdp1 (= MAX FD Plus 1) indicates the number of the maximum descriptor to test, plus one**
  - parameter ignored in Windows sockets
- **returns the number of active descriptors, 0 if terminated for timeout, -1 in case of error**

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfdp1, fd_set *readset,
            fd_set *writeset, fd_set *exceptset,
            struct timeval *timeout);
```

Timeout

- **infinite wait : timeout == NULL**
- **maximum waiting time: timeout != NULL**
- **no waiting (i.e. polling): timeout == {0, 0}**
- **some systems modify the timeout value, consequently it is better to re-initialize it at each call**

```
#include <sys/time.h>
struct timeval
{
  long tv_sec;  // seconds
  long tv_usec; // microseconds
};
```

© A.Lioy - Politecnico di Torino (2003-4)

fd_set

- set of flags to select (file?) descriptors (that is a "bit mask")
- flags are managed with the macro FD_xxx
- FD_ISSET is used to find out which descriptors were modified (sockets used for read or write)
- attention!!! They must be re-initialized at each call

```
#include <sys/select.h>
void FD_ZERO (fd_set *fdset); // reset the mask
void FD_SET (int fd, fd_set *fdset); // set(fd)
void FD_CLR (int fd, fd_set *fdset); // reset(fd)
int  FD_ISSET (int fd, fd_set *fdset); // test(fd)
```

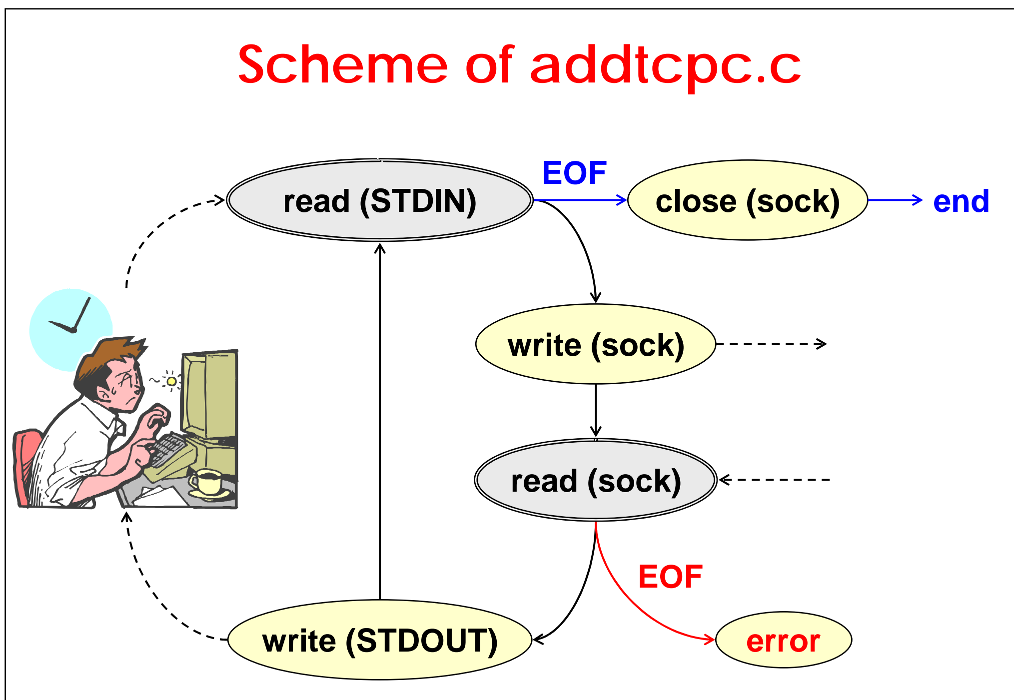
When a descriptor is "ready"?

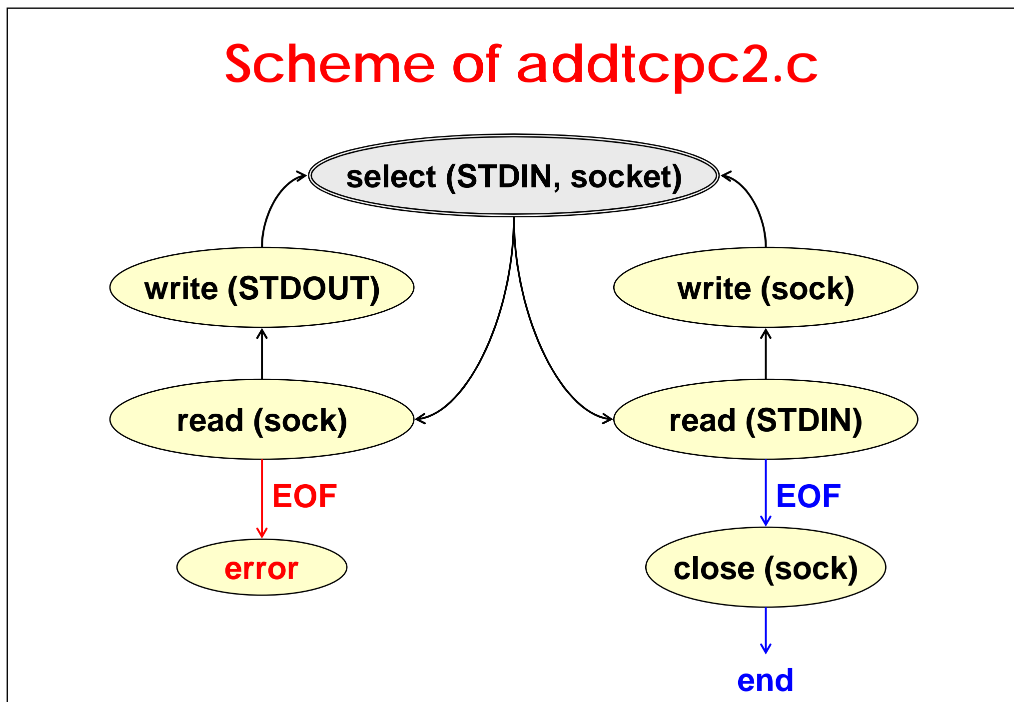
- **readset:**
  - data is available for reading
  - the peer closed the connection for read (that is EOF)
  - there is a new connection to a socket used for read
  - an error on the descriptor appeared
- **writeset:**
  - data can be sent
  - the peer closed the connection for write (SIGPIPE)
  - an error on the descriptor appeared
- **exceptset:**
  - OOB data is available for reading

I/O multiplexing: example

- **modify the client addtcpc.c in such a way to maximize the throughput**
- **solution:**
  - no switching among reading an operation from the standard input and reading the response from the socket but manage both inputs with select( )
  - view the diagrams in the next slides
- **note: make the test either entering input data from the keyboard, either reading it from a file … an error will be noted!**

`addtcpc2.c`



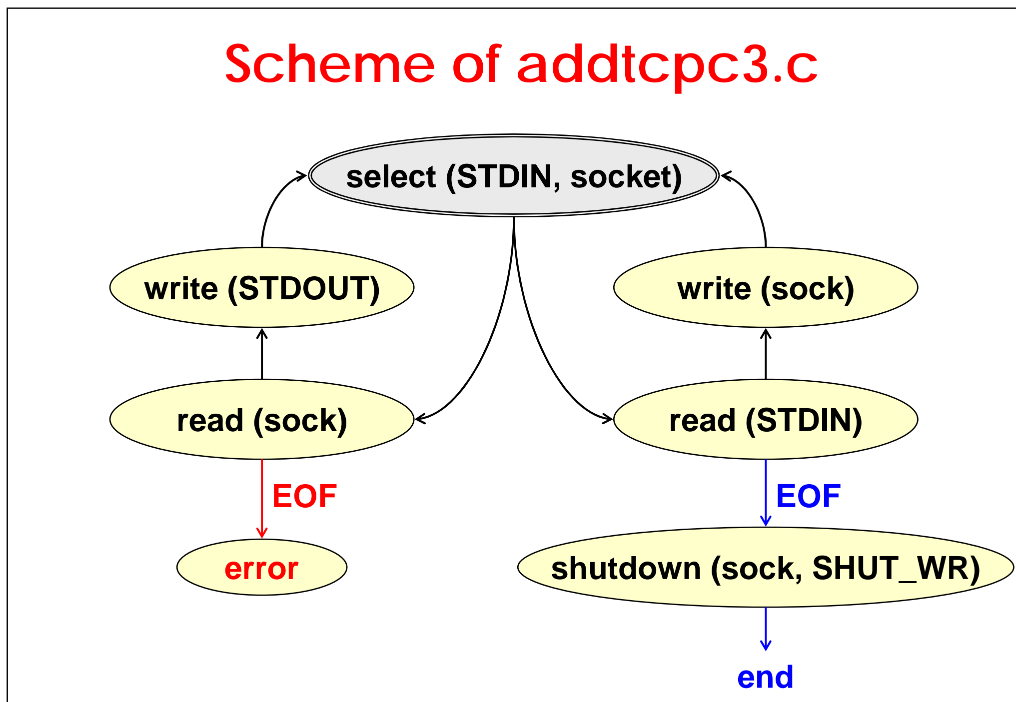


Batch input

- when a flow of continuous input data is provided (like in the case of reading from a file with a large buffer) there exists the risk to terminate all input and close the socket without waiting for all responses
- solution: do not close the socket completely (close) but close only the part for writing (shutdown), waiting to close the part for reading when EOF will be received

`addtcpc3.c`

Scheme of addtcpc3.c



shutdown( )

- **closes one of the two channels associated to a socket**
- **note that close( ):**
  - closes both channels (… but only if the reference count of the descriptor becomes 0)
  - exact behaviour depends on the LINGER option
- **possible values for howto:**
  - SHUT_RD (or 0)
  - SHUT_WR (or 1)
  - SHUT_RDWR (or 2)

```
#include <sys/socket.h>
int shutdown (int sockfd, int howto);
```

shutdown( ) consequences

- **shutdown (sd, SHUT_RD)**
  - impossible to read on socket
  - content of the read buffer is eliminated
  - other data further received are automatically rejected by the stack
- **shutdown (sd, SHUT_WR)**
  - impossible to write on socket
  - content of the write buffer sent to destination, followed by FIN if stream socket

Socket options

getsockopt( ) and setsockopt( )

- cam be applied onky to an open socket
- "level" is the network stack level: SOL_SOCKET, IPPROTO_IP, IPPROTO_TCP, …
- "optname" has mnemonic values

```
#include <sys/socket.h>
#include <netinet/tcp.h>
int getsockopt (int sockfd, int level, int optname,
  void *optval, socklen_t *optlen);
int setsockopt (int sockfd, int level, int optname,
  const void *optval, socklen_t optlen);
```

Some options at the SOCKET level

level	*optname*	*get*	*set*	*type*
SOL_SOCKET	SO_BROADCAST	X	X	int (boolean)
	SO_DEBUG	X	X	int (boolean)
	SO_DONTROUTE	X	X	int (boolean)
	SO_ERROR	X		int
	SO_KEEPALIVE	X	X	int (boolean)
	SO_LINGER	X	X	struct linger
	SO_OOBINLINE	X	X	int (boolean)
	SO_RCVBUF	X	X	int
	SO_SNDBUF	X	X	int
	SO_RCVTIMEO	X	X	struct timeval
	SO_SNDTIMEO	X	X	struct timeval
	SO_REUSEADDR	X	X	int (boolean)
	SO_REUSEPORT	X	X	int (boolean)
	SO_TYPE	X		int

Some options at the IP and TCP levels

level	*optname*	*get*	*set*	*type*
IPPROTO_IP	IP_OPTIONS	X	X	
	IP_TOS	X	X	int
	IP_TTL	X	X	int
	IP_RECVDSTADDR	X	X	int
IPPROTO_TCP	TCP_MAXSEG	X	X	int
	TCP_NODELAY	X	X	int
	TCP_KEEPALIVE	X	X	int

Example: reading socket options

- **UNP, section 7.3**
- **note: in CYGWIN timeouts are integer**

checkopts.c

Broadcast, Keepalive, buffer

- **SO_BROADCAST**
  - only for a datagram socket
  - enables broadcast addresses
- **SO_KEEPALIVE**
  - exchange a “probe” packet every 2 hours (!)
  - value can be changed only at kernel level
- **SO_SNDBUF, SO_RCVBUF**
  - dimension of local buffers; to be set before “connect” (for the client) and “listen” (for the server)
  - value >= 3 x MSS
  - value >= bandwidth x RTT

SO_SNDTIMEO, SO_RCVTIMEO

- **Posix timeouts specified via “struct timeval”**
- **before Posix, timeouts were just an integer**
- **timeouts used only by the following calls:**
  - read, readv, recv, recvfrom, recvmsg
  - write, writev, send, sendto, sendmsg
- **other techniques needed for accept, connect, …**

```
#include <sys/time.h>
struct timeval
{
  long tv_sec;
  long tv_usec;
};
```

SO_REUSEADDR (SO_REUSEPORT)

- **SO_REUSEADDR pertmits:**
  - to bind to a local port locale already occupied by a process (one connection socket and one listen socket)
  - to have more than one server on the same port (es. IP_A, IP_B, INADDR_ANY)
  - to have more than one socket of a single process on the same port but with different local addresses (useful for UDP without IP_RECVDSTADDDR)
  - to have more multicast sockets on the same port (some systems prefer SO_REUSEPORT)
- **this option is highly suggested for all TCP servers**

SO_LINGER

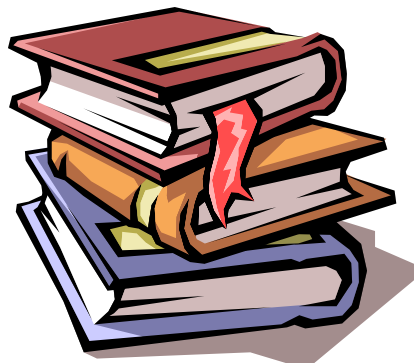
- **changes the behaviour of close( )**
- **OFF: non-blocking close (but attempts to send leftover data)**
- **ON + l_linger == 0: non-blocking close and connection abort (i.e. RST, non FIN + TIME_WAIT)**
- **ON + l_linger > 0: blocking close (attempts to send leftover data until the timeout expires; if fails, then it returns EWOULDBLOCK)**

```
#include <sys/socket.h>
struct linger
{
  int l_onoff; // 0=off, nonzero=on
  int l_linger; // linger timeout (seconds in Posix)
};
```

IP_TOS, IP_TTL

- **read and set the TOS and TTL values frpo output packets**
- **possible TOS values:**
  - IPTOS_LOWDELAY
  - IPTOS_THROUGHPUT
  - IPTOS_RELIABILITY
  - IPTOS_LOWCOST (IPTOS_MINCOST)
- **(theoretical) default values for TTL:**
  - 64 for UDP and TCP sockets (RFC-1700)
  - 255 for RAW sockets

Support libraries

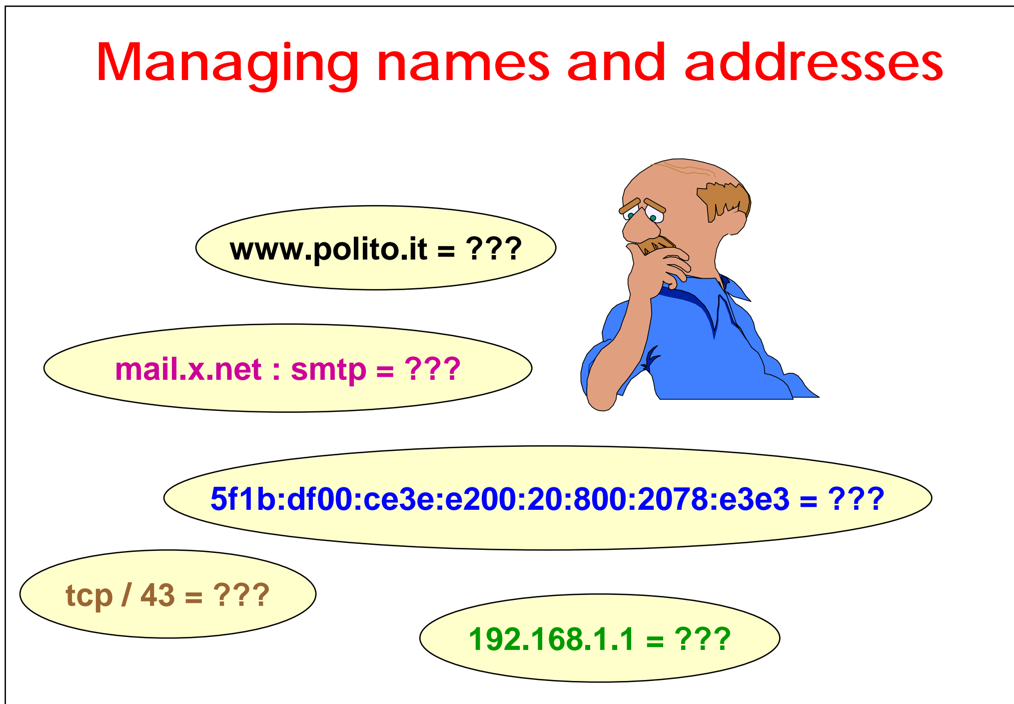


Network libraries

- **libpcap**
  - capture packets
  - http://www.tcpdump.org
  - only for Unix; for Windows see winpcap
- **libdnet**
  - raw IP, raw Ethernet, arp, route, fw, if, addreses
  - http://libdnet.sourceforge.net
  - for Unix and Windows
- **libnet**
  - http://www.packetfactory.net/projects/libnet/

Event libraries

- **libevent**
  - http://monkey.org/~provos/libevent/
- **liboop**
  - http://liboop.org/



Mnemonic and numerical formats

- **conversion among mnemonic names and numerical values (nodes, services, networks, protocols, network addresses)**
- **attention! depends on the local setting of the system**
  - local files (e.g. /etc/hosts, /etc/services)
  - LAN lookup service (e.g. NIS, LDAP)
  - global lookup service (DNS)
- **only in case of DNS it is possible (with some effort) to point explicitly to this service**

gethostbyname( )

- returns a data structure with the description of the node whose name is specified as argument
- in case of error returns NULL and set the variable h_errno to specify the error, for which it can be obtained a textual representation with hstrerror(h_errno)

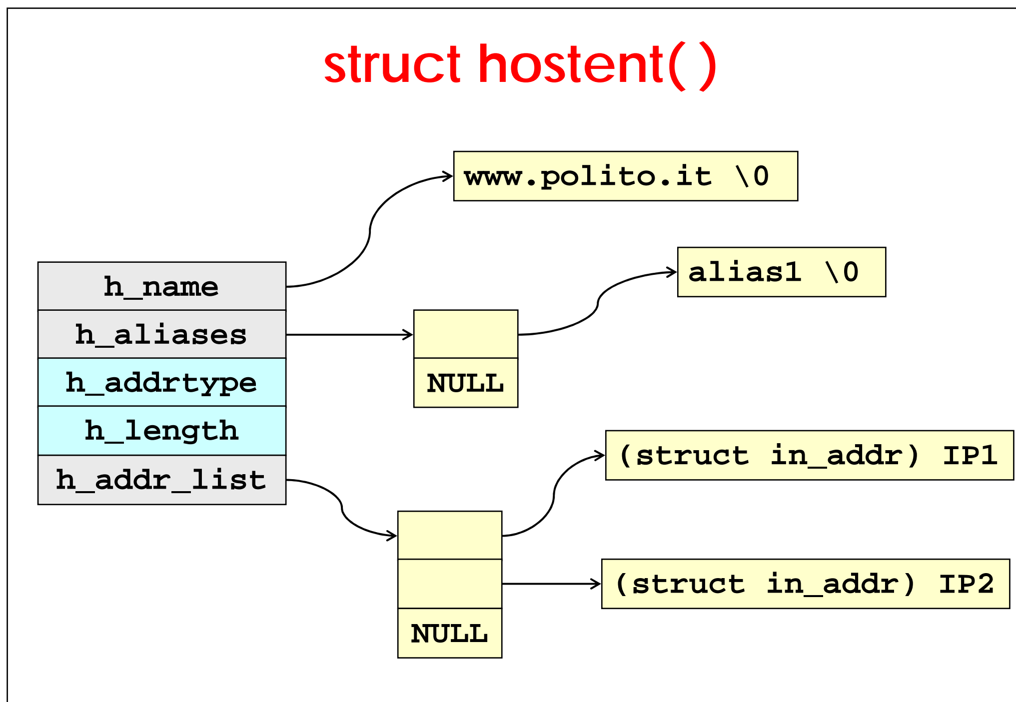
```
#include <netdb.h>
struct hostent *gethostbyname (
  const char *hostname );
extern int h_errno;
char *hstrerror (int h_errno);
```

struct hostent( )

- note: addresses expressed as 1 char = 1 byte (better: 1 unsigned char)

```
#include <netdb.h>
struct hostent {
  char *h_name; // canonical name
  char **h_aliases; // array of alias names
  int  h_addr_type; // AF_INET or AF_INET6
  int  h_length; // address length (4 or 16 bytes)
  char **h_addr_list; // array of addresses
};
#define h_addr h_addr_list[0] // BSD compatibility
```

struct hostent( )



gethostbyaddr( )

- returns a data structure with the description of the node whose address is specified as argument
- in case of error returns NULL and set the variable h_errno to specify the errore, for which it can be obtained a textual representation with hstrerror(h_errno)
- note: in practice the argument addr is a pointer to the struct in_addr or in_addr6

```
#include <netdb.h>
struct hostent *gethostbyaddr (
  const char *addr, size_t len, int family );
```

uname()

- identifies the node on which the program is in execution
- dimension and content of the strings depend on the system and on its configuration
- returns a negative integer in case of error

```
#include <sys/utsname.h>
struct utsname {
  char sysname[...];  // OS name
  char nodename[...]; // network node name
  char release[...];  // OS release
  char version[...];  // OS version
  char machine[...];  // CPU type
};
int uname (struct utsname *nameptr);
```

Examples

- (info_from_n.c) program that provides all the information available about one node given its name
- (info_from_a.c) program that provides all the information available about one node given its address
- (myself.c) program that provides all the information available about one node where the program is running

info_from_n.c
info_from_a.c
myself.c

getservbyname( ), getservbyport( )

- return information on the service whose name or port is given as argument
- return NULL in case of error
- attention: the port numbers are in network order (OK for programming, not for viewing them)

```
#include <netdb.h>
struct servent *getservbyname (
  const char *servname, const char *protoname );
struct servent *getservbyport (
  int port, const char *protoname );
```

struct servent( )

- note : error on UNP (p.251) that defines int s_port

```
#include <netdb.h>
struct servent {
  char   *s_name; // official service name
  char  **s_aliases; // array of aliases
  short   s_port; // port number (network order)
  char   *s_proto; // transport protocol
};
```

service.c

The "daemons"

What is a daemon?

- an autonomous process
- plays typically the role of a server
- activated automatically at boot
- disconnected from any terminal
- write information in a log file
- configured in base of the information passed in in the command line (not advised!) or contained in a configuration file
- executes with privileges of a certain user and group
- executes operations in a certain directory

syslog( )

- is not standard Posix but is mandatory for Unix98
- permits to store data in the log of the system
- actual destination of data depends on the configuration of syslogd (e.g. /etc/syslog.conf)
- communication opened implicitly or explicitly (openlog + closelog)

```
#include <syslog.h>
void syslog (
  int priority, const char *message , ...);
void openlog (
  const char *ident, int options, int facility);
void closelog (void);
```

syslog priority = level | facility

LEVEL

LOG_EMERG	(maximum priority)
LOG_ALERT	
LOG_CRIT	
LOG_ERR	
LOG_WARNING	
LOG_NOTICE	(default priority)
LOG_INFO	
LOG_DEBUG	(minimum priority)

FACILITY

LOG_AUTH	LOG_AUTHPRIV	LOG_CRON
LOG_DAEMON	LOG_FTP	LOG_KERN
LOG_LOCAL0	. . .	LOG_LOCAL7
LOG_LPR	LOG_MAIL	LOG_NEWS
LOG_SYSLOG	LOG_USER	LOG_UUCP

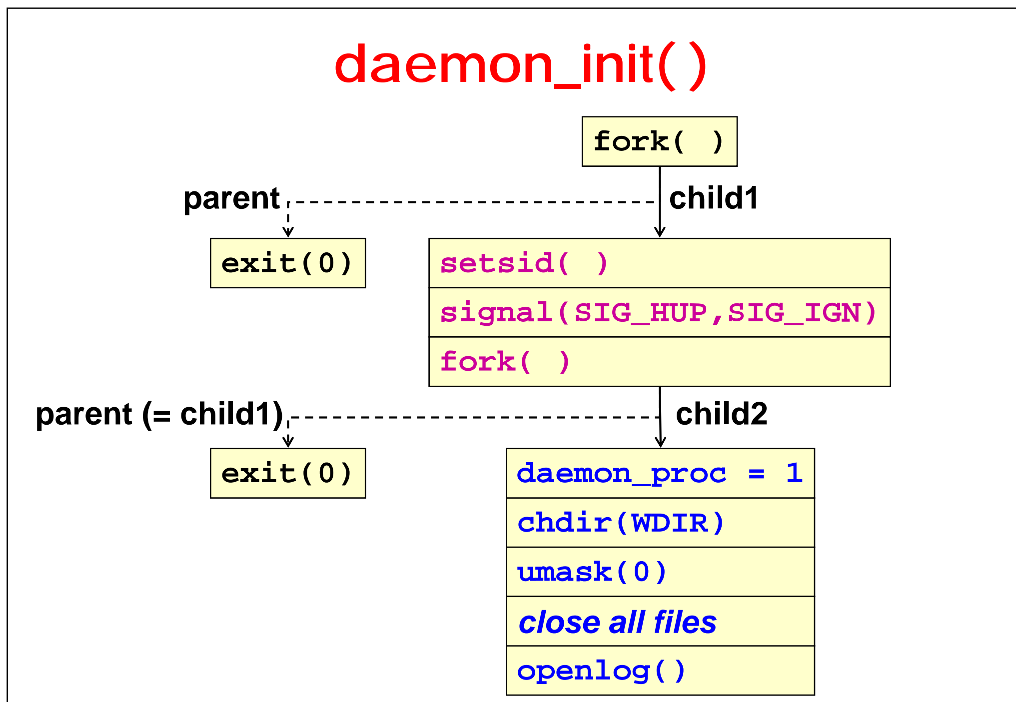
syslog options

OPTIONS

`LOG_CONS`	log on console if the communication with syslogd fails
`LOG_NDELAY`	immediate opening of the socket, without waiting the first call of syslog( )
`LOG_PERROR`	log also on stderr
`LOG_PID`	insert in log also the PID

Initialization of a daemon

- **disconnect from the control terminal (becomes immune to HUP, INT, WINCH that can be used for other signalling)**
- **move in the working directory**
- **close all inherited files**
- **optionally, as caution to averse libraries:**
  - open /dev/null and associte it to stdin, stdout, stderr
  - open a log file and associte it to stdout e stderr
- **open syslog**
- **use the function daemon_init( ) (UNP, p.336) in order not to erroneously initialize a daemon**



Signals towards a daemon

- **signals frequently used:**
  - SIGHUP to read again the configuration file
  - SIGINT to terminate the daemon

inetd

- **if a network node provides several services, it must have more than one daemon listening for service requests, each of which is a process and has source code associated with it**
- **to simplify this case, the super-server "inetd" is often used in Unix**
  - knows the services specified in /etc/inetd.conf
  - listens on all corresponding sockets
  - when it receives a connection request, it activates the corresponding server

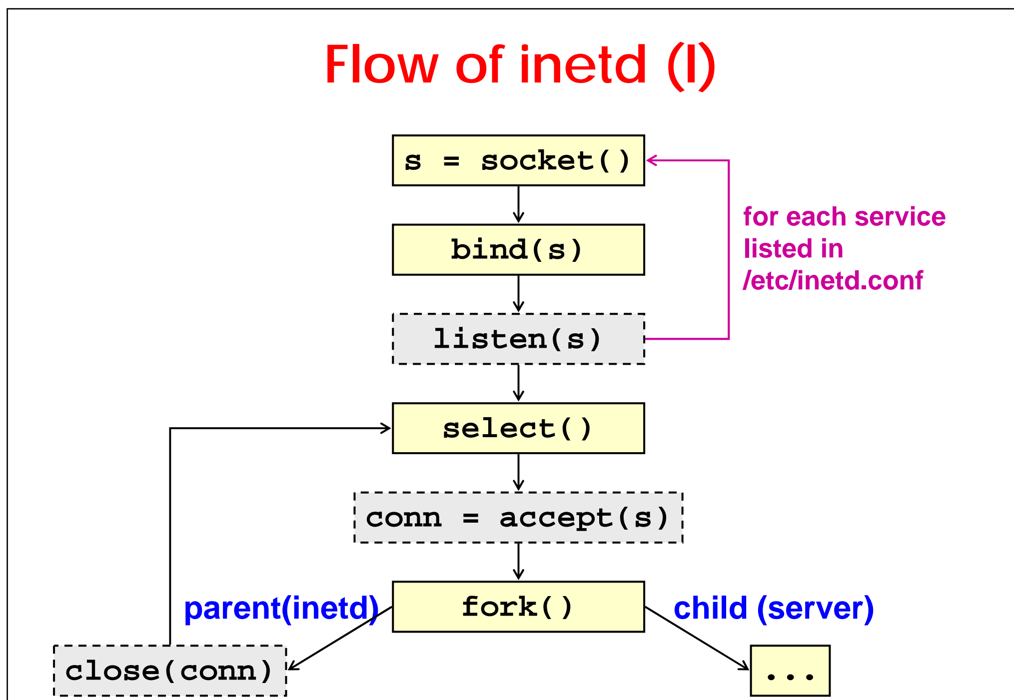
Format of /etc/inetd.conf

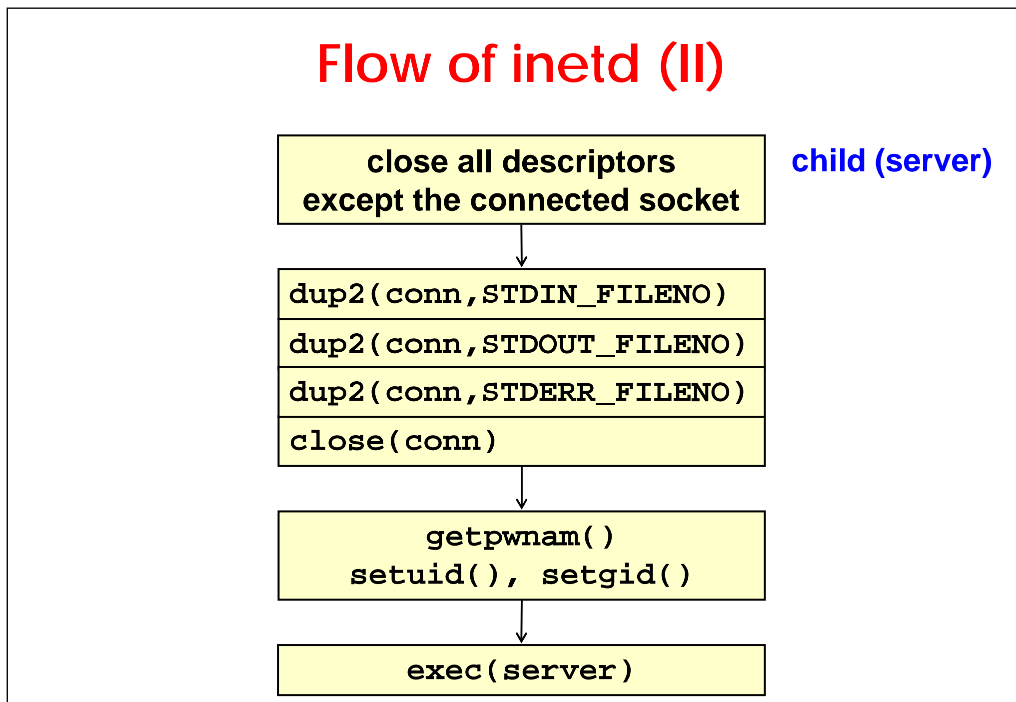
- **each line contains from 7 to 11 fields:**
  - service - name of service in **`/etc/services`**
  - socket type (**`stream`**, **`dgram`**)
  - protocol (**`tcp`**, **`udp`**)
  - flag (**`wait`**, **`nowait`**) - iterative or concurrent server
  - login name (entry in **`/etc/passwd`**) - user name
  - server program (pathname, **`internal`**)
  - arguments - maximum 5, included **`argv[0]`**

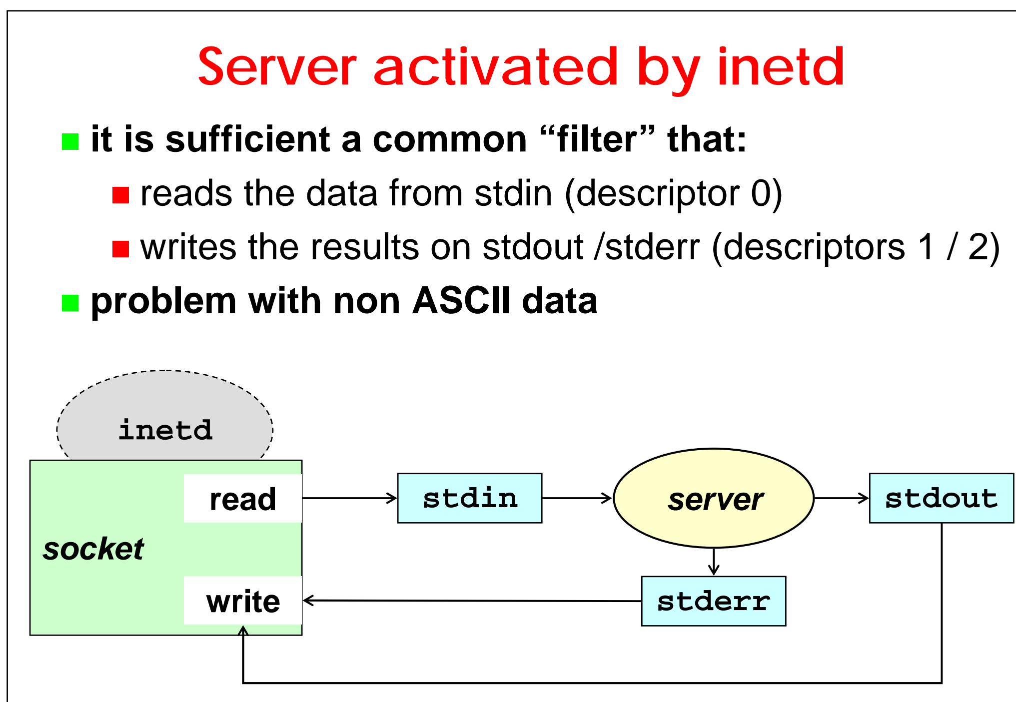
Example of /etc/inetd.conf

```
. . .
echo    stream tcp nowait root   internal
echo    dgram  udp wait   root   internal
ftp     stream tcp nowait root   /usr/etc/ftpd ftpd
telnet  stream tcp nowait root   /usr/etc/telnetd telnetd
login   stream tcp nowait root   /etc/rlogind rlogind
tftp    dgram  udp wait   nobody /usr/etc/tftpd tftpd
talk    dgram  udp wait   root   /etc/talkd talkd
spawn   stream tcp nowait lioy   /usr/local/spawner spawner
. . .
```

Flow of inetd (I)




Flow of inetd (II)
close all descriptors
except the connected socket
child (server)
dup2(conn,STDIN_FILENO)
dup2(conn,STDOUT_FILENO)
dup2(conn,STDERR_FILENO)
close(conn)
getpwnam()
setuid(), setgid()
exec(server)


Server activated by inetd
it is sufficient a common "filter" that:
reads the data from stdin (descriptor 0)
writes the results on stdout /stderr (descriptors 1 / 2)
problem with non ASCII data
inetd
socket
read
write
stdin
server
stdout
stderr

Other solutions for the super-server

- **xinetd is an improved version of inetd**
- **tcpd is a system to activate securely a server from inetd**
- **tcpserver is a complete replacement of inetd+tcpd**