Lab 4

CS-335a

Fall 2012
Computer Science Department

Manolis Surligas
surligas@csd.uoc.gr

Summary

- What is a socket
- POSIX socket API
- Create a TCP server socket
- Create a TCP client socket
- Send and receive data
- Useful man pages

What is sockets?

- Socket is an endpoint of communication between two processes

- Two basic types of sockets:
  - UNIX sockets
  - Network sockets

- Processes read and write data to the sockets in order to communicate

- The reference for socket programming is the POSIX socket API, based on the BSD socket API

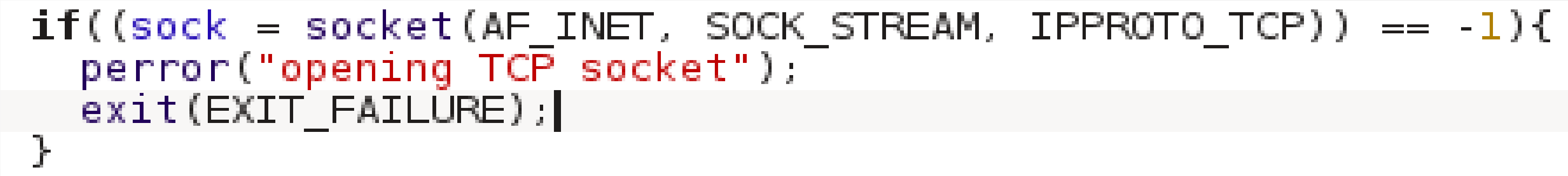
- Based on this API processes use system calls like *socket()*, *bind()*, *send()*, *recv()* to communicate with each other

The POSIX socket API

- Sockets may be of different type and may use different protocols (eg IPv4, IPv6, UNIX, TCP, UDP etc)
- They follow the server-client architecture
- The server creates a socket, and waits until a client connects
- The client creates also a socket and connects to the server
- Reading from a socket results the program to block, until the other process performs a write to its socket
- The opposite is true, **only** for connection oriented sockets (eg TCP sockets)

Open a TCP socket

- Imagine that we want to open an Internet version 4 socket that uses TCP as Transport layer



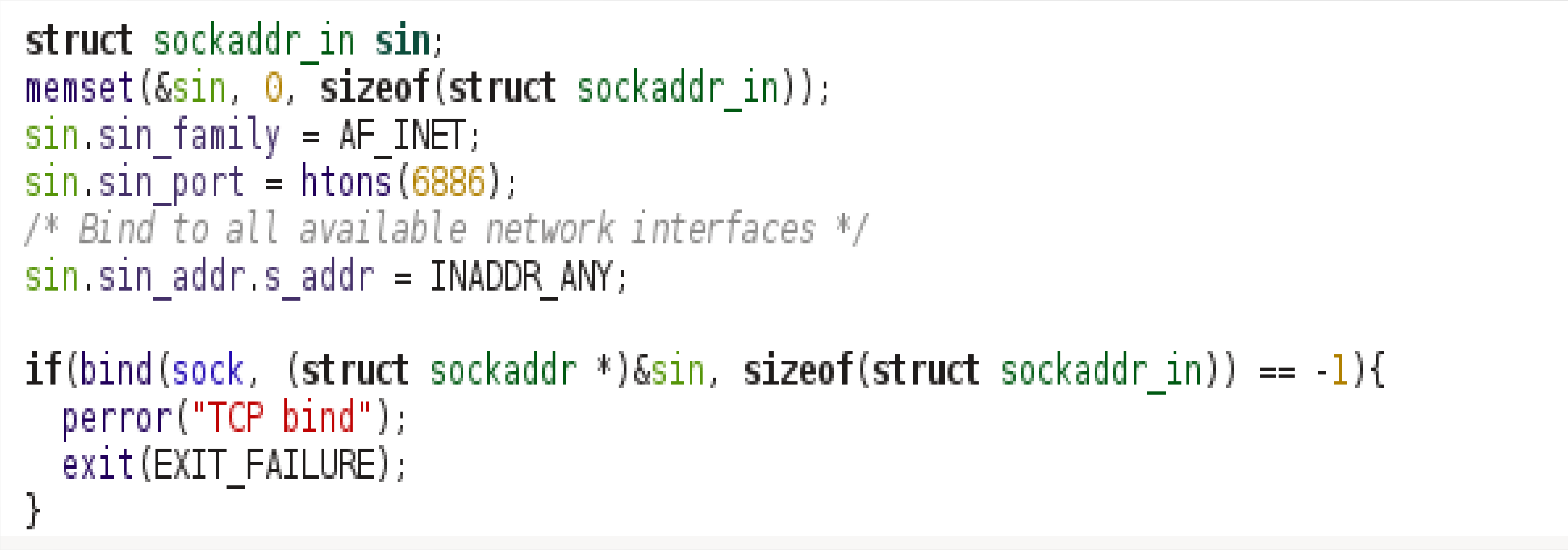
```
if((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1){
  perror("opening TCP socket");
  exit(EXIT_FAILURE);
}
```

  - AF_INET is telling the OS to create an IPv4 socket
  - The type of the socket is the SOCK_STREAM
  - The protocol that we currently use is TCP and this is declared by the IPPROTO_TCP defined value

- Socket() returns the socket descriptor or -1 in case of failure
- Note that we haven't yet specify if this socket is at the server or at the client

Create a TCP Server socket

- At the previous step we created only the socket
- Now we are going to bind this socket with a specific port, in which our server will listen



```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(struct sockaddr_in));
sin.sin_family = AF_INET;
sin.sin_port = htons(6886);
/* Bind to all available network interfaces */
sin.sin_addr.s_addr = INADDR_ANY;

if(bind(sock, (struct sockaddr *)&sin, sizeof(struct sockaddr_in)) == -1){
  perror("TCP bind");
  exit(EXIT_FAILURE);
}
```

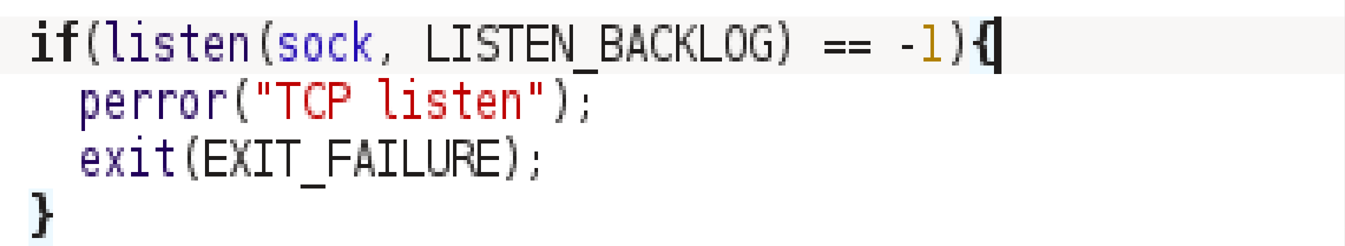
Create a TCP Server socket

- As the family of our socket is AF_INET we use the sockaddr_in structure
- Using memset we initialize all the contents od this structure to 0
- We assign to the *sin_port* the port number that we want our server to listen (function htons will be explained later)
- *sin_addr* field is the address of the interface that we want to bind the socket. INADDR_ANY binds it to all available interfaces
- Again we check if bind() fails. Using perror() we can get the failure reason (eg port already in use)

Create a TCP Server socket

- Are we done? Not yet!

- As a server TCP socket, it should listen at the port that we previously bind it, for incoming connections



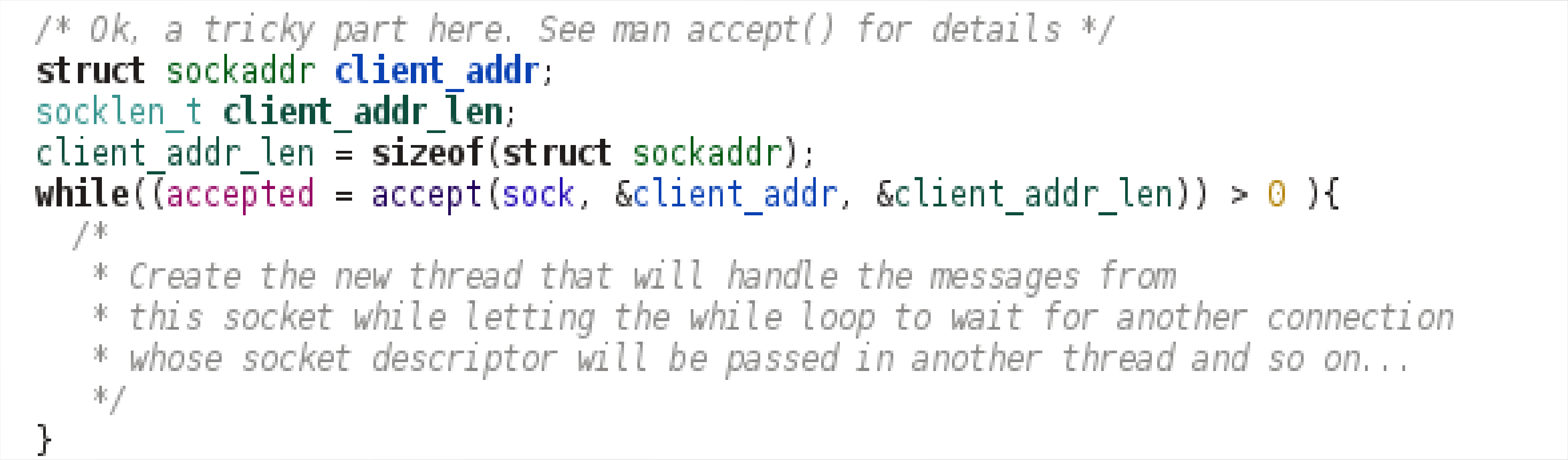
```
if(listen(sock, LISTEN_BACKLOG) == -1){
  perror("TCP listen");
  exit(EXIT_FAILURE);
}
```

- To do that we use the listen() system call

- LISTEN_BACKLOG specifies the maximum number of connections that can wait to the queue before be accepted using *accept()*

Create a TCP Server socket

- We managed to successfully create a listening socket, that can be views with the *netstat -lpt* command
- But how we can handle the incoming connections?



```
/* Ok, a tricky part here. See man accept() for details */
struct sockaddr client_addr;
socklen_t client_addr_len;
client_addr_len = sizeof(struct sockaddr);
while((accepted = accept(sock, &client_addr, &client_addr_len)) > 0 ){
  /*
   * Create the new thread that will handle the messages from
   * this socket while letting the while loop to wait for another connection
   * whose socket descriptor will be passed in another thread and so on...
   */
}
```

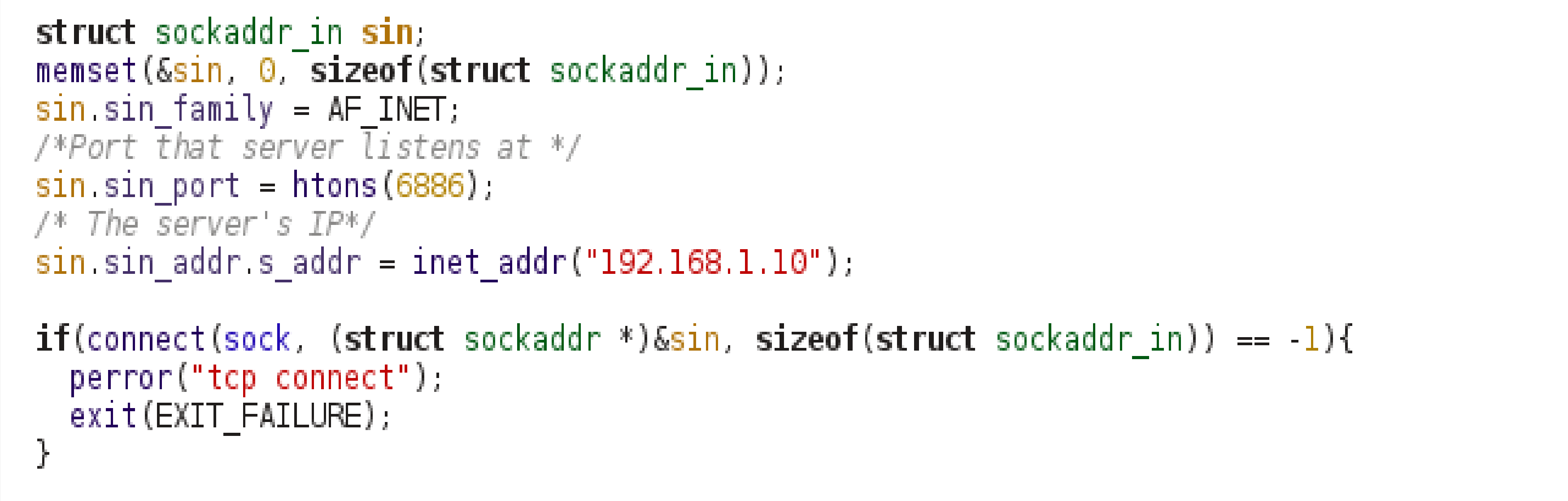
Create a TCP Server socket

- *accept()* until a new connection arrive at the listening socket
- When it does, the incoming connection is assigned to a new socket descriptor and can be handled separately
- The next call of the accept() will get the next available incoming connection
- We this approach we can handle multiple connections at a single server
- Information about the client is stored at the *sockaddr* struct

Create a TCP Client Socket

- At the client side, things are much easier

- After creating a TCP socket, just use connect() to connect with the server at a specific IP address and port number



```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(struct sockaddr_in));
sin.sin_family = AF_INET;
/*Port that server listens at */
sin.sin_port = htons(6886);
/* The server's IP*/
sin.sin_addr.s_addr = inet_addr("192.168.1.10");

if(connect(sock, (struct sockaddr *)&sin, sizeof(struct sockaddr_in)) == -1){
  perror("tcp connect");
  exit(EXIT_FAILURE);
}
```

Send and receive data

- Now both sides are ready to communicate with each other
- The sockets are bidirectional. No need to create one for sending and one for receiving
- Mind the blocking approach. For every write, a read at the other side should be performed
- Note that the blocking behavior can be bypassed with the appropriate arguments
- Use *send()* and *recv()* for sending and receiving data and do **not** make any assumption for the maximum size of a received message

Usefull man pages

- man ip(7)
- man socket(7)
- man socket(3p)
- man bind(3p)
- man listen(3p)
- man accept(3p)
- man connect(3p)
- man send(3p)
- man recv(3p)