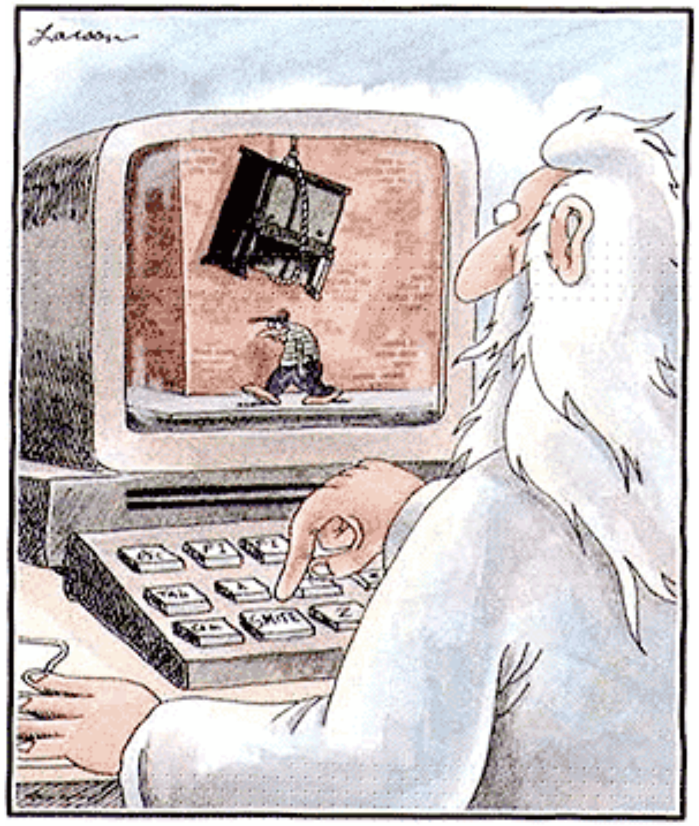
Network Programming Minicourse



God at His computer

TCP Sockets

- Connection-oriented
  - Full two-way data transfer
- Reliable delivery
- Overhead
  - Setup, teardown of connections

UDP Sockets

- Connection-less (datagrams)
- Each packet is independent
- Less overhead, not reliable

Creating Sockets

Important Network Structs

```
struct addrinfo {
   int              ai_flags;
   int              ai_family;
   int              ai_socktype;
   int              ai_protocol;
   size_t           ai_addrlen;
   struct sockaddr *ai_addr;
   char            *ai_canonname;
   struct addrinfo *ai_next;
};
```

Network Structs (cont’d)

```
// all sockaddr structs are cast to this in network calls that
// call for it.
struct sockaddr {
   unsigned short  sa_family; // address family, AF_xxx
   char            sa_data[14]; // 14 bytes of protocol address
};

// used for ipv4.
struct sockaddr_in {
   short           sin_family;
   unsigned short  sin_port;
   struct in_addr  sin_addr;
   char            sin_zero[8];
};
```

getaddrinfo()

```
int getaddrinfo(const char *node,
           const char *service,
           const struct addrinfo *hints,
           struct addrinfo **servinfo);
```

- First three parameters are input params
- Fourth parameter will populated by the function
- Gets you the info you need about the remote end of the connection
- servinfo actually simulates a linked list via the ai_next field, but don't worry about it; just use servinfo directly as a single addrinfo struct

Input Parameters

- node : domain name / ip address
  - "localhost", "127.0.0.1"
- service : port
  - "5555", "1337"
- hints : socket configuration options
  - hints.ai_family: AF_UNSPEC/AF_INET/AF_INET6
  - hints.ai_socktype: SOCK_DGRAM/SOCK_STREAM
    - SOCK_DGRAM is UDP, SOCK_STREAM is TCP
  - hints.ai_flags: ignore this

Results of Function

- int : actual return value
  - used to indicate success / failure
- servinfo : populated by `getaddrinfo()` call
  - contains the info you need to create a socket

socket()

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

- Takes in configuration parameters
  - These will be in servinfo after `getaddrinfo()`
  - Could also hardcode if you really want
- Reserves a file descriptor
  - Used to read/write over the network connection

Input Parameters

- domain
  - AF_INET/AF_INET6
  - use (*servinfo)->ai_family after servinfo is populated by `getaddrinfo()`
- type
  - SOCK_STREAM/SOCK_DGRAM
  - (*servinfo)->ai_socktype
- protocol
  - (*servinfo)->ai_protocol

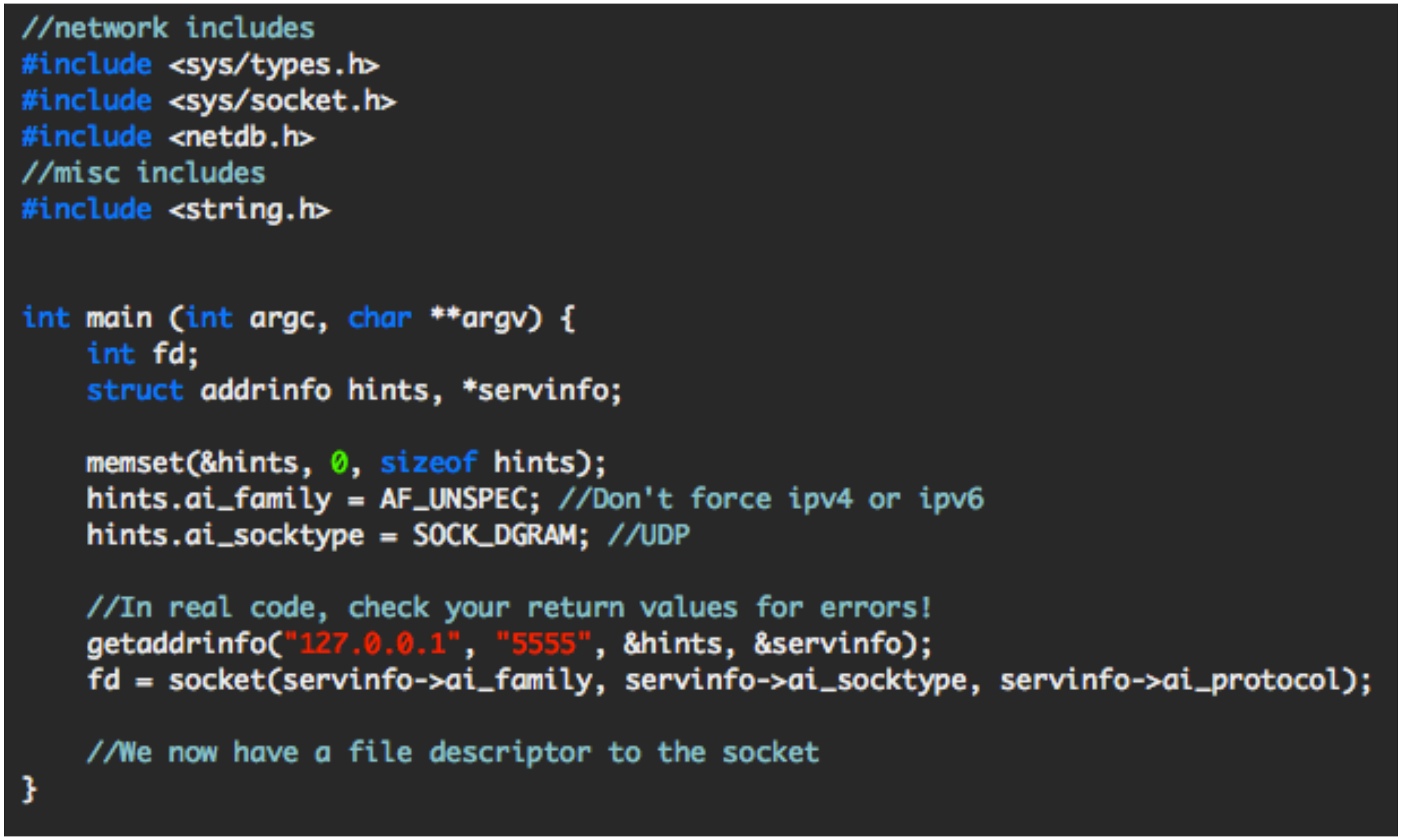
Return Value

- File descriptor for connection
- Same as a file descriptor for a file on disk
- Can read, write, close, shutdown
  - More on these in future slides
- < 0 if error

Quick Caveats (apply to all network functions)

- Remember to error check return values
- Remember to do any necessary validation
- Look at Beej's Network Programming guide for examples of error checking

Some code…



```c
//network includes
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
//misc includes
#include <string.h>


int main (int argc, char **argv) {
    int fd;
    struct addrinfo hints, *servinfo;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; //Don't force ipv4 or ipv6
    hints.ai_socktype = SOCK_DGRAM; //UDP

    //In real code, check your return values for errors!
    getaddrinfo("127.0.0.1", "5555", &hints, &servinfo);
    fd = socket(servinfo->ai_family, servinfo->ai_socktype, servinfo->ai_protocol);

    //We now have a file descriptor to the socket
}
```

Server-side Calls

bind()

```
int bind(int sockfd,
         struct sockaddr *myaddr,
         int addrlen);
```

- Reserves a port to listen on, and specifies its local ip address
  - for our purposes, this ip address will always be localhost (127.0.0.1)
- Connects the socket fd with that reserved port
- Returns < 0 on error

Input Parameters

- sockfd
  - the file descriptor returned by `socket()`
- myaddr
  - the ip address and port to bind to
  - (*servinfo)->ai_addr
- addrlen
  - the length (in bytes) of the ip address
  - (*servinfo)->ai_addrlen

`listen()`

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

- Tells the socket to listen for connections
- Call after `bind()` ties the socket to a port
- Returns < 0 on error

Input Parameters

- sockfd
  - the file descriptor returned from `socket()`
  - same as passed into `bind()`
- backlog
  - maximum number of waiting connections that will be queued
  - not important for our purposes, > 10 is plenty

`accept()`

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

- Called after `listen()`
- Accepts an incoming connection request from a client process
- Probably the trickiest of the socket functions

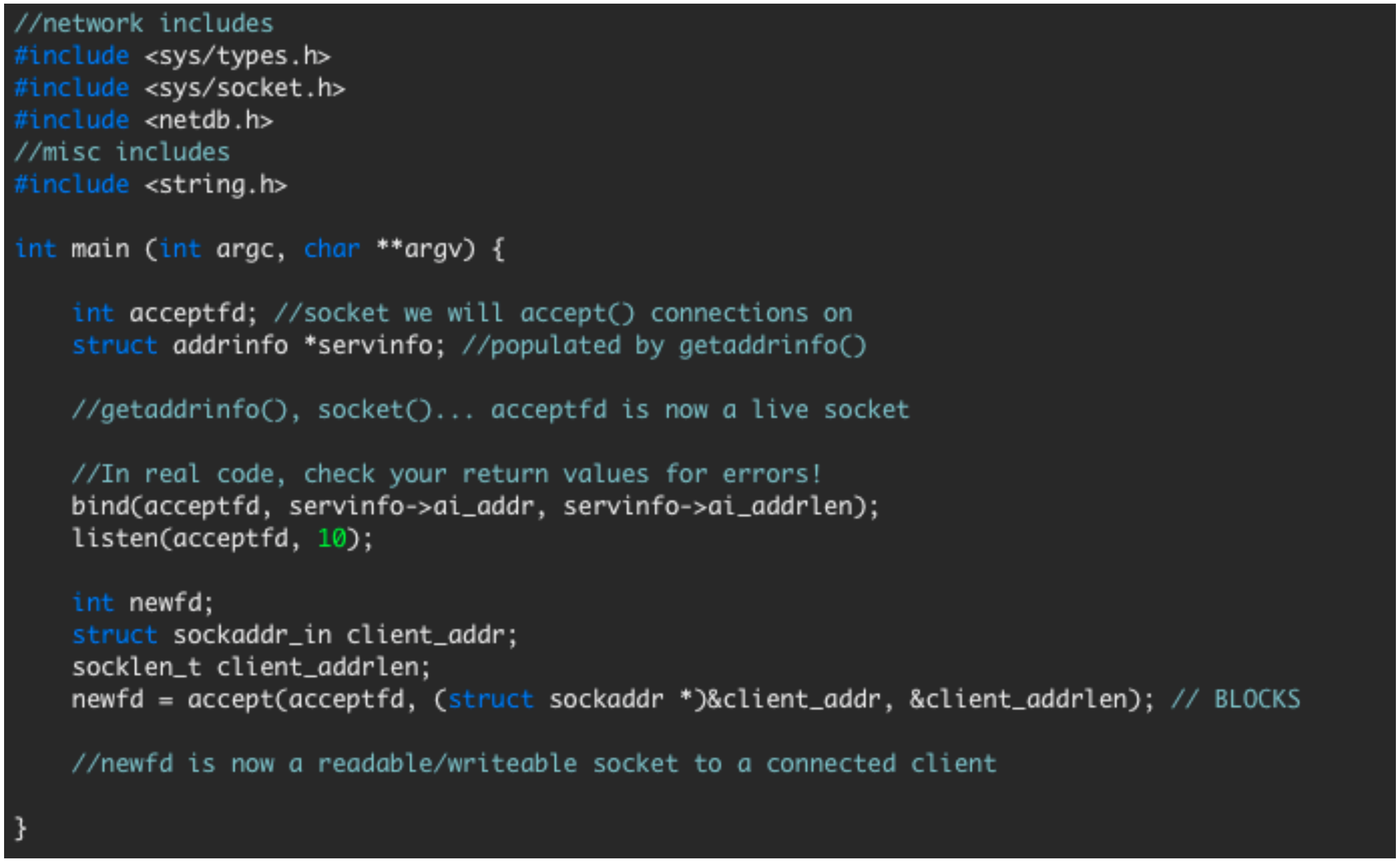
Parameters

- sockfd
  - you guessed it: the same socket fd you used for `bind()` and `listen()`
- addr
  - a pointer to a struct sockaddr
  - will be populated by `accept()`
    - similar to usage of servinfo in `getaddrinfo()`
- addrlen
  - really just points to an integer saying the max size for the client's ip address
  - `accept()` will change the integer if it uses a shorter ip address

`accept()` Usage

- `accept()` blocks!
- When you call `accept()`…
  - Your program blocks, waiting for a client to call `connect()`
    - We'll show you how to use `connect()` in a few slides
- When a client `connect()`s…
  - `accept()` returns a new socket file descriptor for the connection to the client
  - This fd is bound to a random port, and…
  - addr is populated with the info about the other side
  - You don't actually need to use addr, since you can just read/write on the new file descriptor

Code!



```c
//network includes
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
//misc includes
#include <string.h>

int main (int argc, char **argv) {

    int acceptfd; //socket we will accept() connections on
    struct addrinfo *servinfo; //populated by getaddrinfo()

    //getaddrinfo(), socket()... acceptfd is now a live socket

    //In real code, check your return values for errors!
    bind(acceptfd, servinfo->ai_addr, servinfo->ai_addrlen);
    listen(acceptfd, 10);

    int newfd;
    struct sockaddr_in client_addr;
    socklen_t client_addrlen;
    newfd = accept(acceptfd, (struct sockaddr *)&client_addr, &client_addrlen); // BLOCKS

    //newfd is now a readable/writeable socket to a connected client

}
```

Client-side Calls (TCP)

Simpler than server-side!

connect()

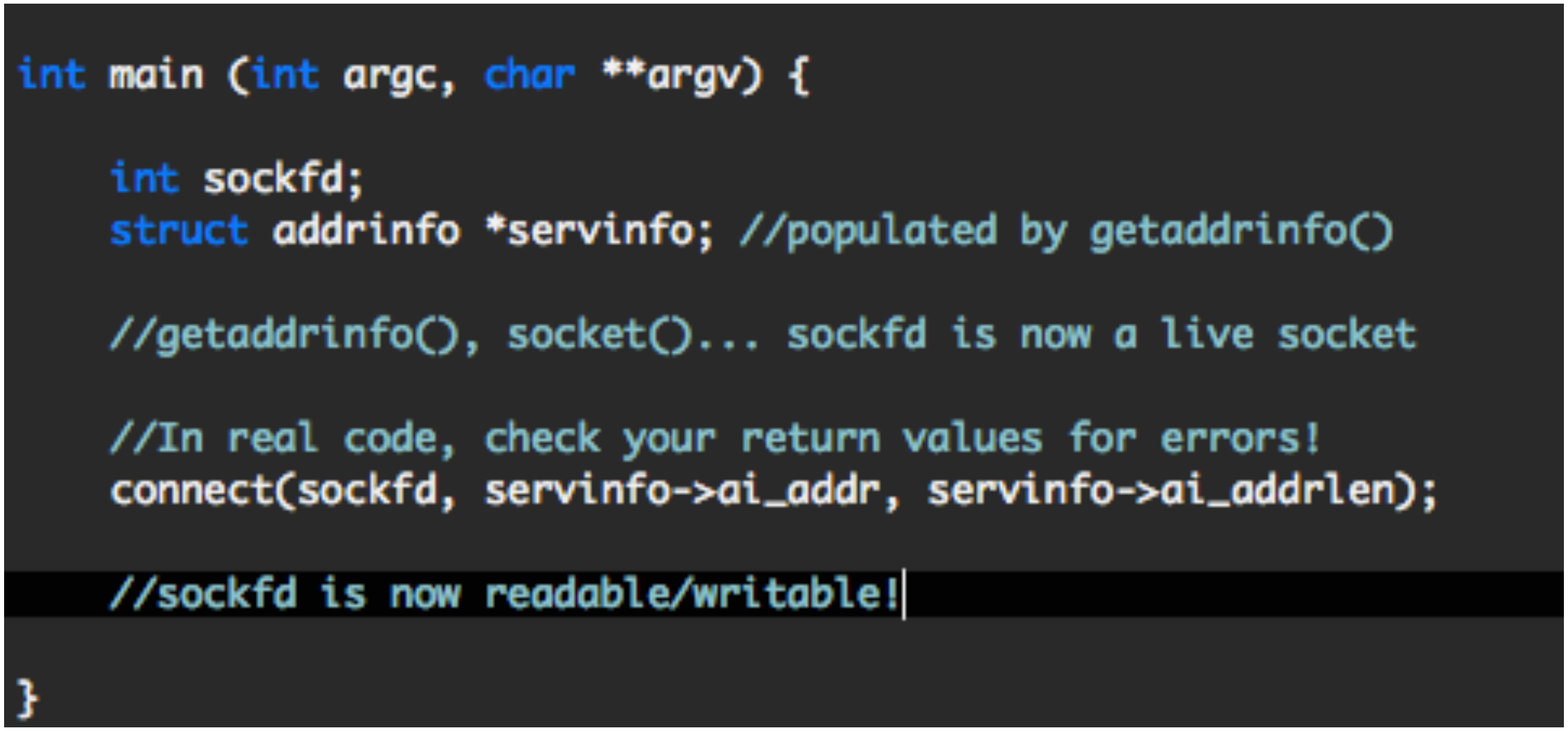
```
int connect(int sockfd,
        struct sockaddr *serv_addr,
        int addrlen);
```

- Connects to a server that is waiting on an `accept()` call
- Returns < 0 on error

Parameters

- sockfd
  - Yup, the file descriptor returned from `socket()`
- serv_addr
  - information about the server to connect to
  - (*servinfo)->ai_addr
- addrlen
  - length (bytes) of server address structure
  - (*servinfo)->ai_addrlen

That’s it! (for the TCP client)



```
int main (int argc, char **argv) {

    int sockfd;
    struct addrinfo *servinfo; //populated by getaddrinfo()

    //getaddrinfo(), socket()... sockfd is now a live socket

    //In real code, check your return values for errors!
    connect(sockfd, servinfo->ai_addr, servinfo->ai_addrlen);

    //sockfd is now readable/writable!


}
```

Sending and Receiving

send()

```
int send(int sockfd,
         const void *msg,
         int len,
         int flags);
```

- use to send data over a stream socket
- returns *the number of bytes actually sent*

Parameters

- sockfd : the socket
- msg
  - a pointer to the data to send
- len
  - length of data to send
- flags
  - don’t worry about this; just set to 0

Return Value

- Number of bytes actually sent
  - -1 on error
- May be less than len!
- For this reason, `send()` needs to be called in a loop to make sure everything is sent

recv()

```
int recv(int sockfd,
         void *buf,
         int len,
         int flags);
```

- For reading over a stream socket
- Blocks until something arrives
- Returns the number of bytes read
  - 0 if connection is remotely closed
  - -1 on error

Parameters

- sockfd
- buf
  - buffer that the data wil be read into
- len
  - maximum length of data to read
  - never set this greater than the size of the buffer
- flags
  - for our purposes, 0

sendto()

```
int sendto(int sockfd,
       const void *msg,
       int len,
       unsigned int flags,
       const struct sockaddr *to,
       socklen_t tolen);
```

- For datagram (UDP) socket sending

Parameters

- First four – exact same as with `send()`
- to
  - remote address to send to
  - (*servinfo)->ai_addr
- tolen
  - length of remote address
  - (*servinfo)->ai_addrlen

Return Value

- Number of bytes sent, 0 on error
- Send in a loop

recvfrom()

```
int recvfrom(int sockfd,
             void *buf,
             int len,
             unsigned int flags,
             struct sockaddr *from,
             int *fromlen);
```

- For datagram (UDP) socket receiving

Parameters

- First four – same as `recv()`
- from
  - `recvfrom()` populates to hold address of sender
- fromlen
  - `recvfrom()` sets to length of “from”

Return Value

- Number of bytes read
  - -1 on error

`close(), shutdown()`

- For killing sockets
- Just use `close(sockfd)`
  - This is just the normal UNIX `close()` call to close a file descriptor

Examples

- There are excellent and comprehensive code examples (as well as explanations of everything here) in Beej's Guide to Network Programming
- http://beej.us/guide/bgnet/

Additional Info

select()

```
int select (int numfds,
               fd_set *readfds,
               fd_set *writefds,
               fd_set *exceptfds,
               struct timeval *timeout);
```

- Informs you when any of a number of sockets have information for reading
- Allows you to monitor a number of connections at once, and even accept new connections, without blocking on any individual one
- Useful to avoid opening a new thread for each connection (hint hint…)

Parameters

- numfds
  - the value of the highest file descriptor plus one
  - keep a running tally
- readfds
  - a set of file descriptors you want to read on
- writefds
  - a set of file descriptors you want to write on
- exceptfds
  - don’t worry about this
- timeout
  - max. time to wait before returning
  - set to NULL to block indefinitely
    - this is probably what you want to do

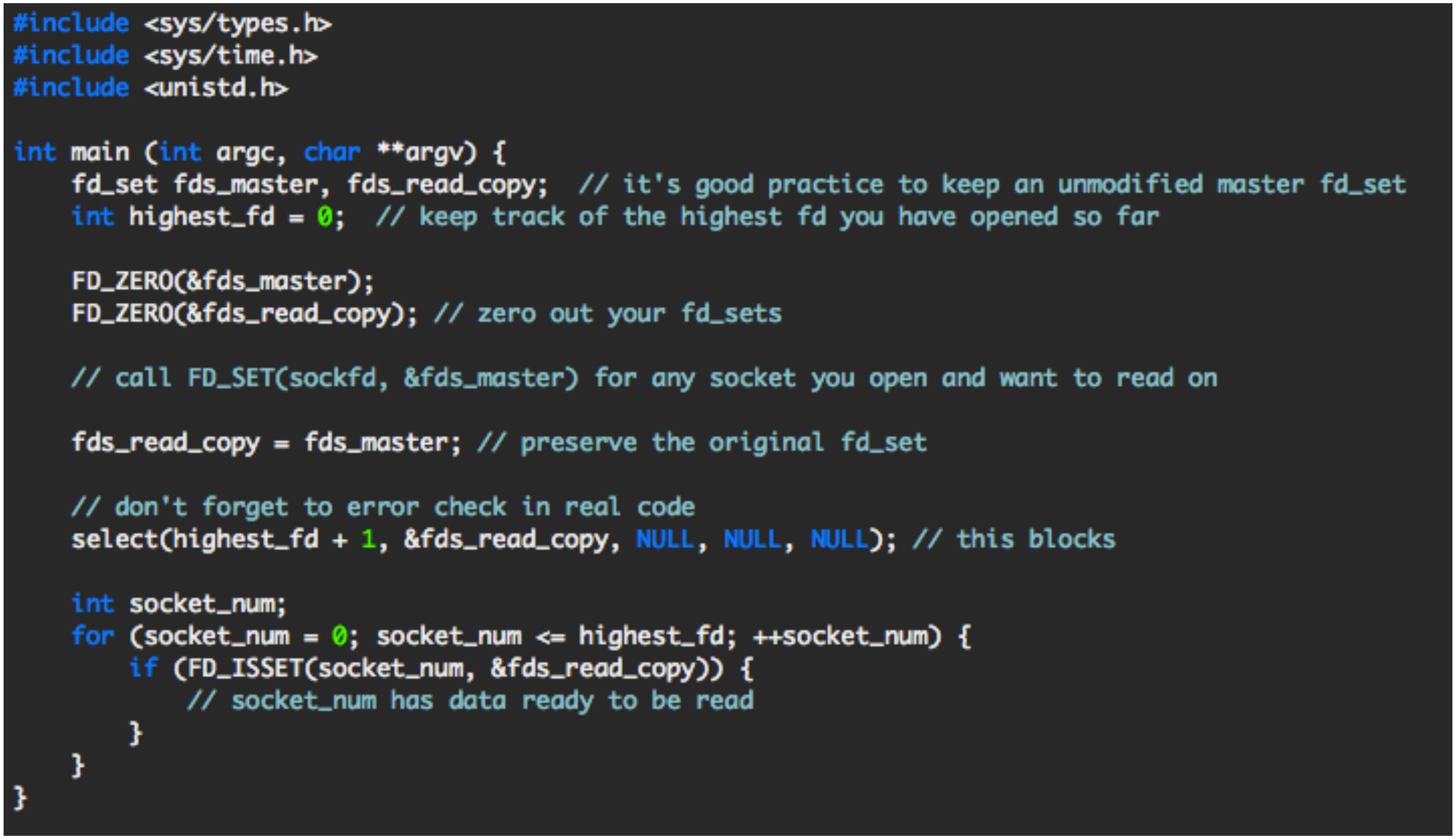
Return Value

- The number of file descriptors ready, or -1 on error

Useful Macros

- Used for managing fd_sets
- `FD_SET(int fd, fd_set *set)`
  - Add an fd to an fd_set
- `FD_CLR(int fd, fd_set *set)`
  - Remove an fd from an fd_set
- `FD_ISSET(int fd, fd_set *set)`
  - Check whether fd is set
  - Used once `select()` returns to see if the fd is ready for reading / writing
- `FD_ZERO(fd_set *set)`
  - Clear an fd_set

So how do I use `select()`?



```c
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

int main (int argc, char **argv) {
    fd_set fds_master, fds_read_copy;  // it's good practice to keep an unmodified master fd_set
    int highest_fd = 0;  // keep track of the highest fd you have opened so far

    FD_ZERO(&fds_master);
    FD_ZERO(&fds_read_copy); // zero out your fd_sets

    // call FD_SET(sockfd, &fds_master) for any socket you open and want to read on

    fds_read_copy = fds_master; // preserve the original fd_set

    // don't forget to error check in real code
    select(highest_fd + 1, &fds_read_copy, NULL, NULL, NULL); // this blocks

    int socket_num;
    for (socket_num = 0; socket_num <= highest_fd; ++socket_num) {
        if (FD_ISSET(socket_num, &fds_read_copy)) {
            // socket_num has data ready to be read
        }
    }
}
```

Tips

- Add your `accept()`-ing fd to the fd_set you `select()` on
  - that way you don’t have to block on `accept()`

Byte Order

- `ntohs(), ntohl(), htons(), htonl()`
- "network to host short", "network to host long", "host to network short", "host to network long"
- Makes sure that all info is sent over the wire in the same byte order
- Call `hton()` before sending data over the wire
- Call `ntoh()` after reading data off of the wire