CSE 333

Lecture 8 - file and network I/O

Steve Gribble

Department of Computer Science & Engineering

University of Washington

Administrivia

HW1 was due yesterday

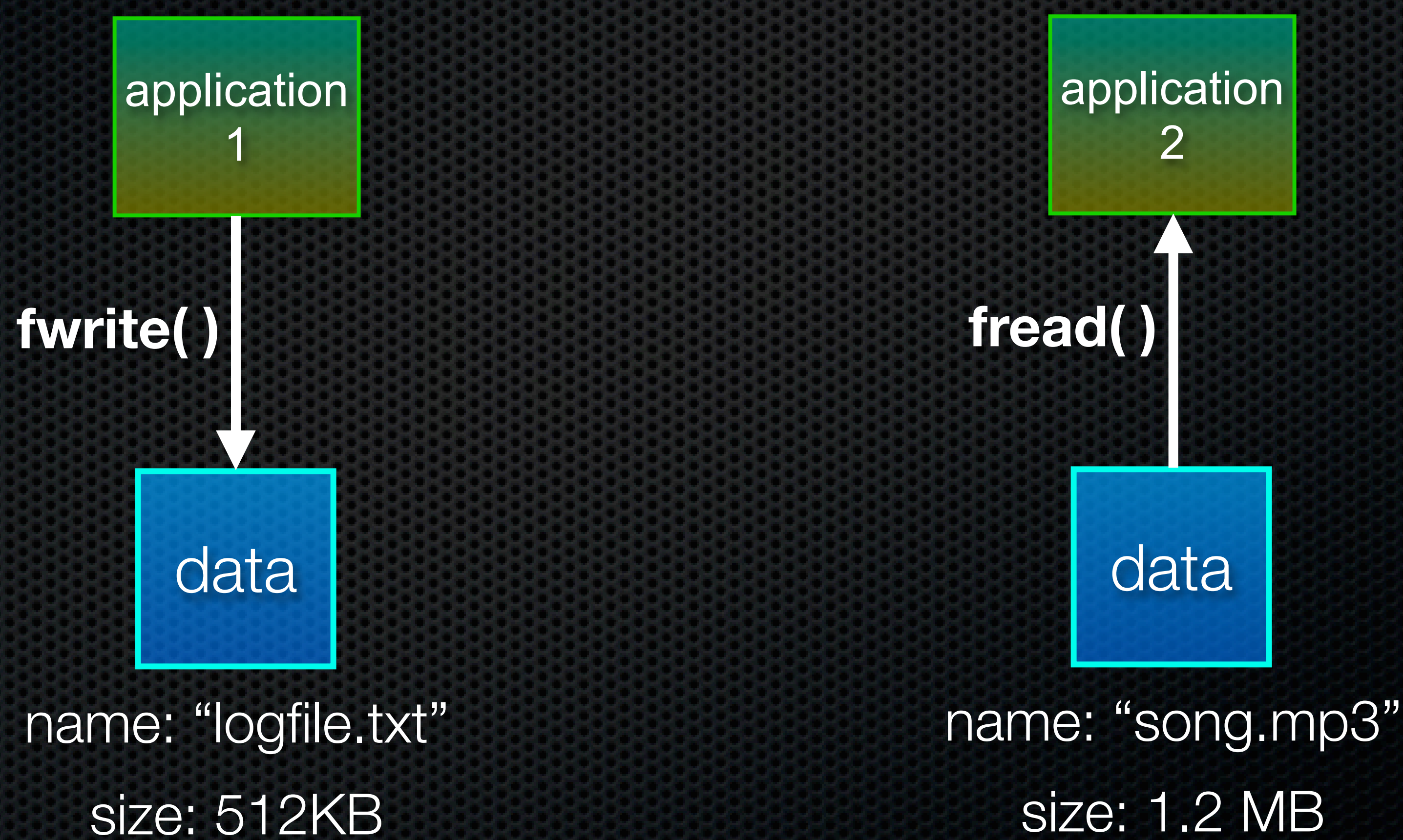
- phew! how'd it go?

HW2 will go out on Monday

- builds on HW1
- we'll provide libhw1.a as an alternative if you find that your solution is buggy

Files

File I/O


application
1
fwrite( )
data
name: “logfile.txt”
size: 512KB
application
2
fread( )
data
name: “song.mp3”
size: 1.2 MB

Files

Simple semantics

- files are named by filename
  - [directory name] / [filename]
  - e.g., /Users/gribble/files/song.mp3
- files have a current size
  - finite
  - discoverable
- programs typically read from, or write to, a file
  - occasionally both (e.g., database)

Let's do some file I/O...

We'll start by using C's standard library

- functions are defined within libC
- they make use of Linux system calls

C's **stdio** defines the notion of a **stream**

- a stream is a way of reading or writing a sequence of characters from/to a device
  - a stream can be either *text* or *binary;* Linux does not distinguish
  - a stream is *buffered* by default; libc reads ahead of you
  - three streams are provided by default: **stdin**, **stdout**, **stderr**
  - you can open additional streams to read/write to files

Using C streams

fread_example.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define READBUFSIZE 128
int main(int argc, char **argv) {
  FILE *f;
  char readbuf[READBUFSIZE];
  size_t readlen;

  if (argc != 2) {
    fprintf(stderr, "usage: ./fread_example filename\n");
    return EXIT_FAILURE;  // defined in stdlib.h
  }

  // Open, read, and print the file
  f = fopen(argv[1], "rb");  // "rb" --> read, binary mode
  if (f == NULL) {
    fprintf(stderr, "%s -- ", argv[1]);
    perror("fopen failed -- ");
    return EXIT_FAILURE;
  }

  // Read from the file, write to stdout.
  while ((readlen = fread(readbuf, 1, READBUFSIZE, f)) > 0)
    fwrite(readbuf, 1, readlen, stdout);
  fclose(f);
  return EXIT_SUCCESS;  // defined in stdlib.h
}
```

printf(...) is equivalent to fprintf(stdout, ...)

stderr is a stream for printing error output to a console

fopen opens a stream to read or write a file

perror writes a string describing the last error to stderr

stdout is for printing non-error output to the console

Writing is easy too

see cp_example.c

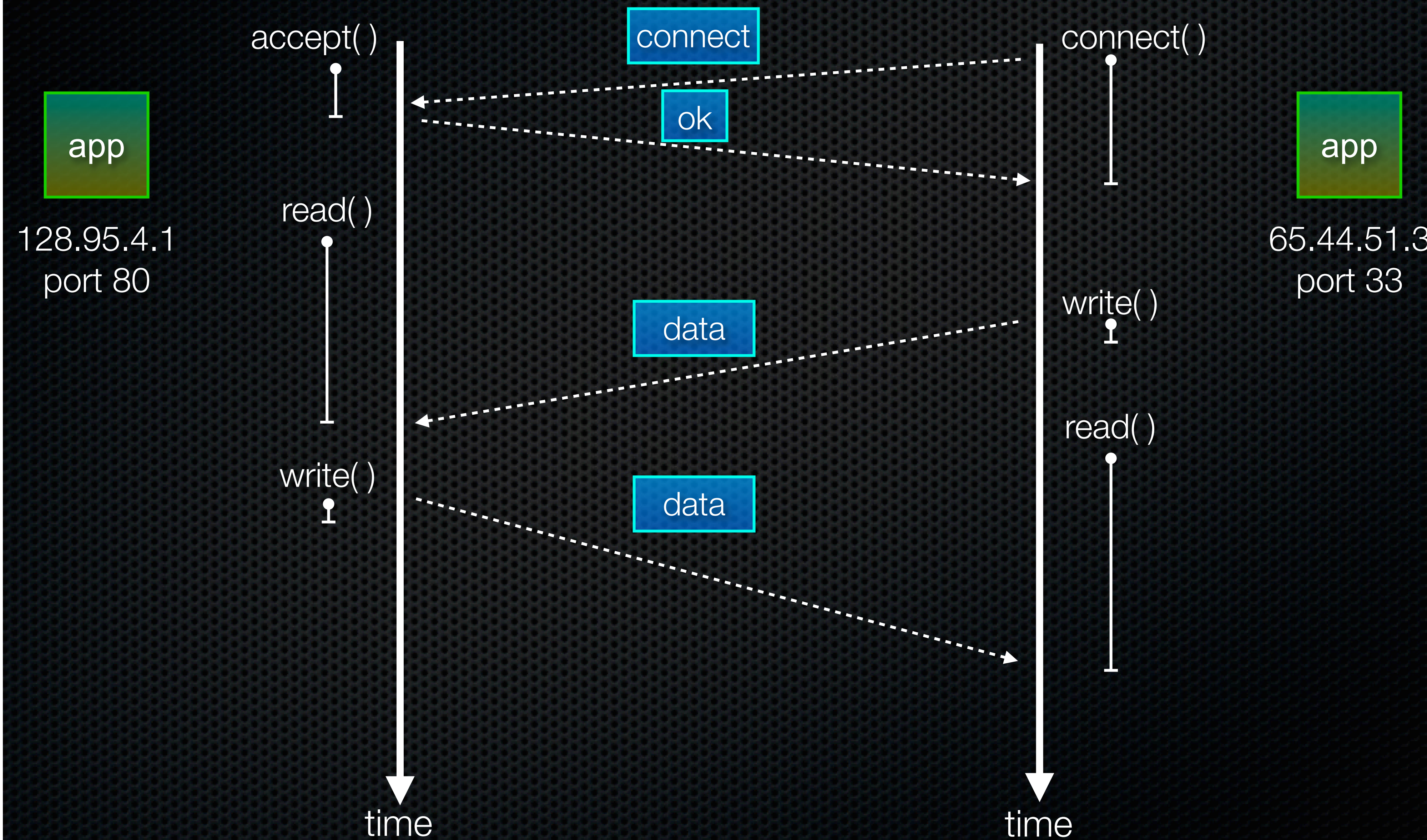
Files and file descriptors

The OS provides open, read, write, and close

- system calls for interacting with files
- open( ) returns a *file descriptor*
  - an integer that represents an open file
  - inside the OS, it's an index into a table that keeps track of any state associated with your interactions, such as the file position
  - you pass the file descriptor into read, write, and close
- read( ) and write( ) are unbuffered

Network

Network I/O



accept( )
connect
connect( )
ok
app
app
read( )
128.95.4.1
port 80
65.44.51.3
port 33
write( )
data
read( )
write( )
data
time
time

Network I/O

Complex semantics

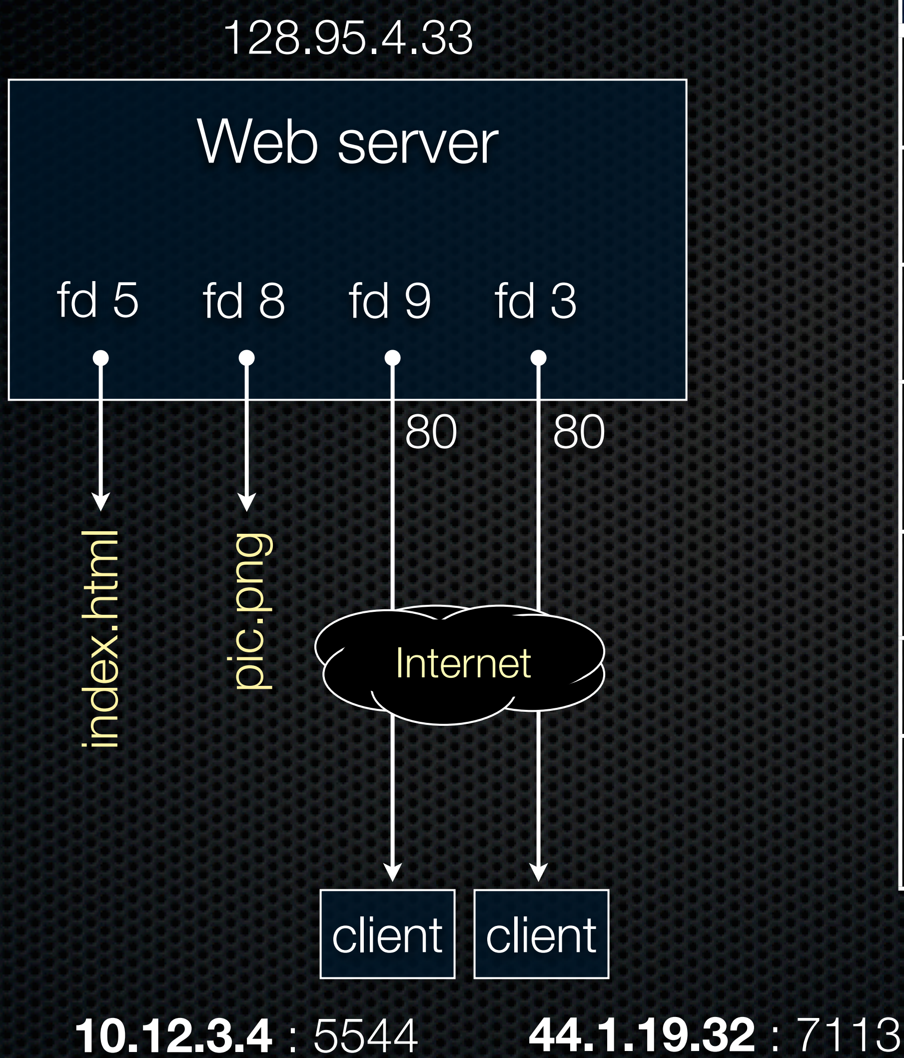
- endpoints named by IP address / port number
  - e.g., IP address 128.95.4.1,   port 80
- communication is bidirectional
  - each participant interleaves reading and writing data
  - a protocol dictates **who** sends **what**, and **when**
- an endpoint can't predict how much data will arrive, or when it will arrive
  - fread( ) isn't really appropriate, since it will block until a specified amount of data has been read, or EOF is reached
  - but EOF for network means "other side hung up"

Network I/O

read( ) is more appropriate than fread( )

- read(int fd, void *buf, size_t bufsize)
  - you specify "I'm willing to receive **bufsize** bytes"
  - read will block in OS until some data is available
  - once data is available, read will return up to **bufsize** bytes
    - will return less than bufsize if less data is available
- lets you block until something can be read
  - without knowing how much you should read

Pictorially



OS's descriptor table

file descriptor	type	connected to?
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

Types of sockets

Stream sockets

- for connection-oriented, point-to-point, reliable bytestreams
  - uses TCP, SCTP, or other stream transports

Datagram sockets

- for connection-less, one-to-many, unreliable packets
  - uses UDP or other packet transports

Stream sockets ("TCP")

Typically used for client / server communications

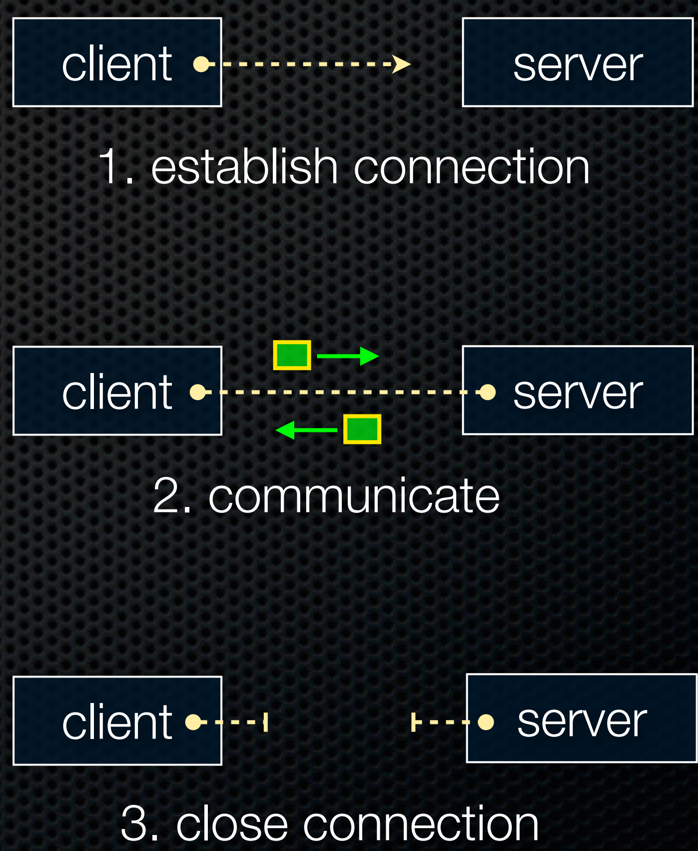
- but also for other architectures, like peer-to-peer

Client

- an application that establishes a connection to a server

Server

- an application that receives connections from clients



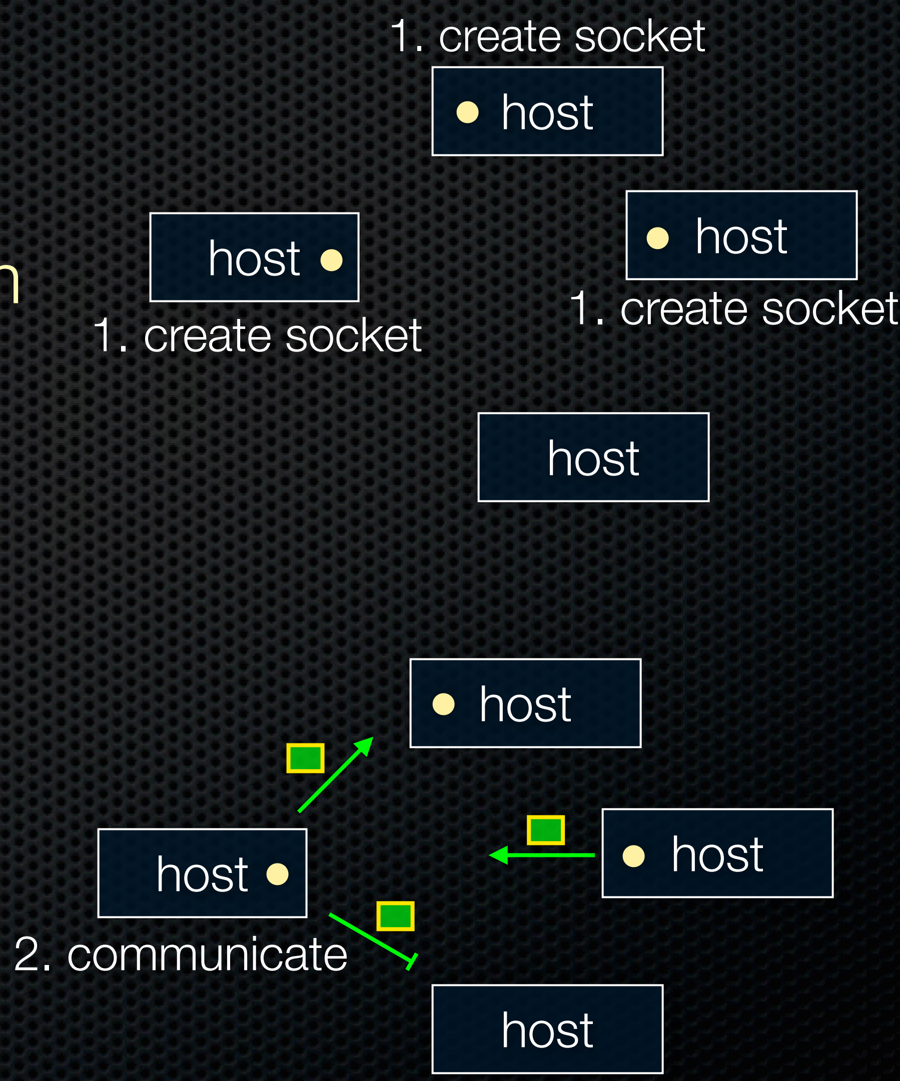
Datagram sockets (“UDP”)

Used less frequently than stream sockets

- they provide no flow control, ordering, or reliability

Often used as a building block

- streaming media applications
- sometimes, DNS lookups



The sockets API

Berkeley sockets originated in 4.2 BSD Unix circa 1983

- it is the standard API for network programming
  - available on most OSs

POSIX socket API

- a slight updating of the Berkeley sockets API
  - a few functions were deprecated or replaced
  - better support for multi-threading was added

Let’s dive into it!

We’ll start by looking at the API from the point of view of a client connecting to a server over TCP

- there are five steps:
  1. figure out the address and port to which to connect
  2. create a socket
  3. connect the socket to the remote server
  4. read and write data using the socket
  5. close the socket

Connecting from a client to a server.

Step 1. Figure out the address and port to which to connect.

Network addresses

For IPv4, an IP address is a 4-byte tuple

- e.g., 128.95.4.1 (80:5f:04:01 in hex)

For IPv6, an IP address is a 16-byte tuple

- e.g., 2d01:0db8:f188:0000:0000:0000:0000:1f33
    - 2d01:0db8:f188::1f33 in shorthand

IPv4 address structures

```
// Port numbers and addresses are in *network order*.

// A mostly-protocol-independent address structure.
struct sockaddr {
    short int     sa_family;    // Address family; AF_INET, AF_INET6
    char          sa_data[14];  // 14 bytes of protocol address
};

// An IPv4 specific address structure.
struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET == IPv4
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Padding

struct in_addr {
    uint32_t s_addr;  // IPv4 address
};
```

IPv6 address structures

```
// A structure big enough to hold either IPv4 or IPv6 structures.
struct sockaddr_storage {
    sa_family_t  ss_family;      // address family

    // a bunch of padding; safe to ignore it.
    char         __ss_pad1[_SS_PAD1SIZE];
    int64_t      __ss_align;
    char         __ss_pad2[_SS_PAD2SIZE];
};

// An IPv6 specific address structure.
struct sockaddr_in6 {
    u_int16_t        sin6_family;   // address family, AF_INET6
    u_int16_t        sin6_port;     // Port number
    u_int32_t        sin6_flowinfo; // IPv6 flow information
    struct in6_addr  sin6_addr;     // IPv6 address
    u_int32_t        sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char    s6_addr[16];   // IPv6 address
};
```

Generating these structures

Often you have a string representation of an address

- how do you generate one of the address structures?

genaddr.cc

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

int main(int argc, char **argv) {
  struct sockaddr_in sa; // IPv4
  struct sockaddr_in6 sa6; // IPv6

  // IPv4 string to sockaddr_in.
  inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));

  // IPv6 string to sockaddr_in6.
  inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));

  return EXIT_SUCCESS;
}
```

Generating these structures

How about going in reverse?

genstring.cc

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

int main(int argc, char **argv) {
  struct sockaddr_in6 sa6;          // IPv6
  char astring[INET6_ADDRSTRLEN];   // IPv6

  // IPv6 string to sockaddr_in6.
  inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));

  // sockaddr_in6 to IPv6 string.
  inet_ntop(AF_INET6, &(sa6.sin6_addr), astring, INET6_ADDRSTRLEN);
  printf("%s", astring);

  return EXIT_SUCCESS;
}
```

DNS

People tend to use DNS names, not IP addresses

- the sockets API lets you convert between the two
- it’s a complicated process, though:
  - a given DNS name can have many IP addresses
  - many different DNS names can map to the same IP address
    - an IP address will reverse map into at most one DNS names, and maybe none
  - a DNS lookup may require interacting with many DNS servers

You can use the “dig” Linux program to explore DNS

- “man dig”

Resolving DNS names

The POSIX way is to use **getaddrinfo( )**

- a pretty complicated system call; the basic idea...
  - set up a “hints” structure with constraints you want respected
    - e.g., IPv6, IPv4, or either
  - tell getaddrinfo( ) which host and port you want resolved
    - host: a string representation; DNS name or IP address
  - getaddrinfo( ) returns a list of results packed in an “addrinfo” struct
  - free the addrinfo structure using freeaddrinfo( )

DNS lookup example

see dnsresolve.cc

Connecting from a client to a server.

Step 2. Create a socket.

Creating a socket

Use the **socket** system call

- creating a socket doesn't yet bind it to a local address or port

socket.cc

```
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <iostream>

int main(int argc, char **argv) {
  int socket_fd = socket(PF_INET, SOCK_STREAM, 0);
  if (socket_fd == -1) {
     std::cerr << strerror(errno) << std::endl;
     return EXIT_FAILURE;
  }
  close(socket_fd);
  return EXIT_SUCCESS;
}
```

Connecting from a client to a server.

Step 3. Connect the socket to the remote server.

connect( )

The **connect( )** system call establishes a connection to a remote host

- you pass the following arguments to connect( ):
  - the socket file descriptor you created in step 2
  - one of the address structures you created in step 1
- connect may take some time to return
  - it is a **blocking** call by default
  - the network stack within the OS will communicate with the remote host to establish a TCP connection to it
  - this involves ~2 *round trips* across the network

connect example

see connect.cc

Connecting from a client to a server.

Step 4. read and write data using the socket.

read( )

By default, a blocking call

- if there is data that has already been received by the network stack, then read will return immediately with it
  - as mentioned before, read might return with less data than you asked for
- if there is no data waiting for you, by default read( ) will block until some arrives
  - then will return with whatever data has arrived

write( )

By default, also a blocking call

- but, in a more sneaky way
- when write( ) returns, the receiver (i.e., the other end of the connection) probably has not yet received the data
  - in fact, the data might not have been sent on the network yet!
  - write( ) enqueues your data in an OS “send buffer” and then returns
  - the OS will transmit the data in the background
- if there is no more space left in the send buffer, by default write( ) will block

read and write are complex

If you call read, you will block until data is available

- but, for esoteric reasons, read( ) might return early
    - indicates this by returning -1 (i.e., an error) and setting the global variable errno to EINTR
    - the right thing to do is try your read again
- so, you have to put read in a while loop to handle this case

read/write example

see sendreceive.cc

Connecting from a client to a server.

Step 5. close( ) the socket.

See you on Wednesday!

Exercise 1

Write a program that:

- uses argc/argv to receive the name of a text file
- reads the contents of the file a line at a time
- parses each line, converting text into a uint32_t
- builds an array of the parsed uint32_t's
- sorts the array
- prints the sorted array to stdout
  - hints: use "man" to read about getline, sscanf, realloc, and qsort

```
bash$ cat in.txt
1213
3231
000005
52
bash$ ex1 in.txt
5
52
1213
3231
bash$
```

Exercise 2

Write a program that:

- loops forever; in each loop, it:
  - prompts the user to input a filename
  - reads from stdin to receive a filename
  - opens and reads the file, and prints its contents to stdout, in the format shown on the right
- hints:
  - use "man" to read about fgets
  - or if you're more courageous, try "man 3 readline" to learn about libreadline.a, and google to learn how to link to it

```
0000000 50 4b 03 04 14 00 00 00 00 00 9c 45 26 3c f1 d5
0000010 68 95 25 1b 00 00 25 1b 00 00 0d 00 00 00 43 53
0000020 45 6c 6f 67 6f 2d 31 2e 70 6e 67 89 50 4e 47 0d
0000030 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 91 00
0000040 00 00 91 08 06 00 00 00 c3 d8 5a 23 00 00 00 09
0000050 70 48 59 73 00 00 0b 13 00 00 0b 13 01 00 9a 9c
0000060 18 00 00 0a 4f 69 43 43 50 50 68 6f 74 6f 73 68
0000070 6f 70 20 49 43 43 20 70 72 6f 66 69 6c 65 00 00
0000080 78 da 9d 53 67 54 53 e9 16 3d f7 de f4 42 4b 88
0000090 80 94 4b 6f 52 15 08 20 52 42 8b 80 14 91 26 2a
00000a0 21 09 10 4a 88 21 a1 d9 15 51 c1 11 45 45 04 1b
00000b0 c8 a0 88 03 8e 8e 80 8c 15 51 2c 0c 8a 0a d8 07
00000c0 e4 21 a2 8e 83 a3 88 8a ca fb e1 7b a3 6b d6 bc
...etc.
```

Exercise 3

Write a program that:

- connects to the hostname and port provided by argv[1] and argv[2]
- reads data from stdin into memory
  - once stdin has closed (you hit EOF), writes the data to the socket
  - once the socket has closed, quits