Client-side Networking
CSE 333 Spring 2019

Instructor: Justin Hsia

Teaching Assistants:
Aaron Johnston   Andrew Hu   Daniel Snitkovskiy
Forrest Timour   Kevin Bi     Kory Watson
Pat Kosakanchit  Renshu Gu   Tarkan Al-Kazily
Travis McGaha
Adminstrivia

- hw3 is due Thursday (5/23)
  - Usual reminders: don’t forget to tag, clone elsewhere, and recompile
- hw4 out on Friday (5/24)
- Exercise 15 will be released on Thursday
  - Client-side TCP connection
  - Related to section this week
Resolving DNS Names

- The POSIX way is to use `getaddrinfo()`
  - A complicated system call found in `#include <netdb.h>`

```c
int getaddrinfo(const char* hostname,
                const char* service,
                const struct addrinfo* hints,
                struct addrinfo** res);
```

- Tell `getaddrinfo()` which host and port you want resolved
  - String representation for host: DNS name or IP address
- Set up a “hints” structure with constraints you want respected
- `getaddrinfo()` gives you a list of results packed into an “addrinfo” structure/linked list
  - Returns 0 on success; returns negative number on failure
- Free the `struct addrinfo` later using `freeaddrinfo()` recursively frees `res` linked list
getaddrinfo

- getaddrinfo() arguments:
  - hostname – domain name or IP address string
  - service – port # (e.g. "80") or service name (e.g. "www")
    or NULL/nullptr

- struct addrinfo {
  int ai_flags; // additional flags
  int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
  int ai_socktype; // SOCK_STREAM, SOCK_DGRAM, 0
  int ai_protocol; // IPPROTO_TCP, IPPROTO_UDP, 0
  size_t ai_addrlen; // length of socket addr in bytes
  struct sockaddr* ai_addr; // pointer to socket addr
  char* ai_canonname; // canonical name
  struct addrinfo* ai_next; // can form a linked list
};
DNS Lookup Procedure

1) Create a `struct addrinfo` hints
2) Zero out `hints` for “defaults”
3) Set specific fields of `hints` as desired
4) Call `getaddrinfo()` using `&hints`
5) Resulting linked list `res` will have all fields appropriately set

- See `dnsresolver.cc`
Socket API: Client TCP Connection

- There are five steps:
  1) Figure out the IP address and port to connect to
  2) Create a socket
  3) Connect the socket to the remote server
  4) `read()` and `write()` data using the socket
  5) Close the socket
Step 2: Creating a Socket

- **int socket(int domain, int type, int protocol);**
  - Creating a socket doesn’t bind it to a local address or port yet
  - Returns file descriptor or -1 on error

```cpp
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <iostream>

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

socket.cc
Step 3: Connect to the Server

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

  ```c
  int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
  ```

  - `sockfd`: Socket file description from Step 2
  - `addr` and `addrlen`: Usually from one of the address structures returned by `getaddrinfo` in Step 1 (DNS lookup)
  - Returns 0 on success and -1 on error

- `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`

```c++
// Get an appropriate sockaddr structure.
struct sockaddr_storage addr;
size_t addrlen;
LookupName(argv[1], port, &addr, &addrlen);

// Create the socket.
int socket_fd = socket(addr.ss_family, SOCK_STREAM, 0);
if (socket_fd == -1) {
    cerr << "socket() failed: " << strerror(errno) << endl;
    return EXIT_FAILURE;
}

// Connect the socket to the remote host.
int res = connect(socket_fd,
                   reinterpret_cast<sockaddr*>(&addr),
                   addrlen);
if (res == -1) {
    cerr << "connect() failed: " << strerror(errno) << endl;
}
```
Review Question

- How do we error check `read()` and `write()`?

A. `ferror()` for `fread()`/`fwrite()`

B. Return value less than expected

C. Return value of 0 or NULL

D. Return value of -1

E. We’re lost…
Step 4: read()

- If there is data that has already been received by the network stack, then read will return immediately with it
  - read() might return with less data than you asked for

- If there is no data waiting for you, by default read() will block until something arrives
  - How might this cause deadlock?
  - Can read() return 0?
    - Yes, if connection is closed
Step 4: read()

- Assume we have:
  - `int socket_fd;` // fd of connected socket
  - `char readbuf[BUF];` // read buffer
  - `int res;` // to store read result

- Write C++ code to read in BUF characters from socket_fd
  - If error occurs, send error message to user and `exit()`

```cpp
while(1) {
    res = read(socket_fd, readbuf, BUF);
    if (res == -1) {
        if (errno == EINTR)
            continue;
        std::cerr << "read error: " << strerror(errno) << std::endl;
        close(socket_fd);
        exit(EXIT_FAILURE);
    }
    ...
}
```

See `sendreceive.cc` for complete code.
Step 4: `write()`

- `write()` queues your data in a send buffer in the OS and then returns
  - The OS transmits the data over the network in the background
  - When `write()` returns, the receiver probably has not yet received the data!

- If there is no more space left in the send buffer, by default `write()` will block
Read/Write Example

- See `sendreceive.cc`

```c
while (1) {
    int wres = write(socket_fd, readbuf, res);
    if (wres == 0) {
        cerr << "socket closed prematurely" << endl;
        close(socket_fd);
        return EXIT_FAILURE;
    } else if (wres == -1) {
        if (errno == EINTR)
            continue;
        cerr << "socket write failure: " << strerror(errno) << endl;
        close(socket_fd);
        return EXIT_FAILURE;
    }
    break;
}
```
Step 5: `close()`

- `int close(int fd);`

  - Nothing special here – it’s the same function as with file I/O
  - Shuts down the socket and frees resources and file descriptors associated with it on both ends of the connection
Extra Exercise #1

- Write a program that:
  - Reads DNS names, one per line, from stdin
  - Translates each name to one or more IP addresses
  - Prints out each IP address to stdout, one per line