Announcements

- **Assignment 4 Due Tonight, Assignment 5 Out Today**
- **Today's Agenda**
    - Work through Ice Cream Parlor simulation example and wrap up threading!
    - Time permitting, we'll launch into our next big topic: networking!
        - We'll learn how the file descriptor concept is extended to allow data (text, HTML, images, JavaScript) to be read from anywhere—not just to and from local files, from **`stdin`**, or to **`stdout`** and **`stderr`**.
        - We'll study how to write scalable, networked applications using the Berkeley and POSIX sockets API.
    - Reading:
        - Read Sections 4.1 and 4.2 of the Saltzer & Kaashoek textbook. These two sections provide a wonderful discussion of the client-server model.
        - Read all of Chapter 11 of Bryant & O'Hallaron (which is the third chapter of your reader.)

Hostname Resolution

- **`gethostbyname`** and **`gethostbyaddr`**
  - Linux C includes directives to convert host names (e.g. "www.facebook.com") to IP address (e.g. "31.13.75.17") and vice versa. Functions called **`gethostbyname`** and **`gethostbyaddr`**, while technically deprecated, are still so prevalent that you should know how to use them. In fact, your B&O textbook only mentions these deprecated functions:

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

  - Each function populates a statically allocated **`struct hostent`** describing some host machine on the Internet.
    - **`gethostbyname`** assumes argument is a host name, as with **`www.google.com`**.
    - **`gethostbyaddr`** assumes the first argument is a binary representation of an IP address (e.g. not the string "171.64.64.137", but the base address of a character array with ASCII values of 171, 64, 64, and 137 laid down side by side in *network byte order*. The second argument is usually 4 for IPv4 addresses (the ones we're familiar with), but could be more for IPv6 addresses.
      The third argument is generally **`AF_INET`** (for IPv4 addresses), but might be **`AF_INET6`** (for IPv6 addresses) or specify some other address family. We'll rely exclusively on **`AF_INET`** in CS110.

Hostname Resolution

- **`struct hostent`**
  - The **`struct hostent`** record packages all of the information about a particular host contributing to the Internet.

```
struct hostent {
  char *h_name;         // official name of host
  char **h_aliases;     // NULL-terminated list of aliases
  int h_addrtype;       // host address type, e.g. AF_INET
  int h_length;         // length of address (4 for IPv4 addresses)
  char **h_addr_list;   // NULL-terminated list of IP addresses
}; // h_addr_list is really a struct in_addr ** when known to be IPv4 addresses

struct in_addr {
  unsigned int s_addr  // stored in network byte order (big endian)
};
```

Resolving IP Addresses

- **`gethostbyname` often used in network applications.**
  - This shouldn't surprise you, as users prefer the host naming scheme behind "www.facebook.com", but network communication ultimately works with binary representation of "31.13.75.17".
  - Here's the core of the full program that queries the user for hostnames and uses **`gethostbyname`** to surface information about them:

```
static void publishIPAddressInfo(const string& host) {
  struct hostent *he = gethostbyname(host.c_str());
  if (he == NULL) { // NULL return value means resolution attempt failed
    cout << host << " could not be resolved to an address." << endl;
    return;
  }

  cout << "Official name is \"" << he->h_name << "\"" << endl;
  cout << "IP Addresses: " << endl;
  struct in_addr **addressList = (struct in_addr **) he->h_addr_list;
  while (*addressList != NULL) {
    cout << "+ " << inet_ntoa(**addressList) << endl;
    addressList++;
  }
}
```

  - Note two implementation features:
    - **`h_addr_list`** is typed to be a **`char *`** array and implies it's an array of C strings, even dotted quad IP addresses.
      - That's not correct. **`h_addr_list`** is really an array of **`struct in_addr *`**s.
      - Each **`in_addr`** is a gratutitous **`struct`** wrapper around an unsigned int, which is just big enough to store the four bytes of an IP address. These four bytes are stored in network byte order (e.g. big endian order)
    - The **`inet_ntoa`** accepts **`struct in_addr`**s and returns their dotted quad equivalents (e.g. "171.45.34.199") as statically allocated C strings.

Resolving IP Addresses: Take II

- **Technically, `gethostbyname` and `gethostbyaddr` are deprecated.**
  - We're *supposed* to use **`getaddrinfo`**, **`getnameinfo`**, and **`freeaddrinfo`** instead.
    - But they're more complicated, because they expose the concept of a socket address and the various data structures used to represent them.
  - Here are the prototypes of the functions we need to be concerned with:

```
int getaddrinfo(const char *hostname, const char *servname,
                const struct addrinfo *hints, struct addrinfo **res);
void freeaddrinfo(struct addrinfo *ai);
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen, int flags);
```

  - **`getaddrinfo`**
    - **`getaddrinfo`** accepts the name of the host as the first argument.
    - The second argument can be something like "80" or "http" if you'd like to programmatically filter away IP addresses that aren't configured to act as a web server, etc. For our purposes, we'll pass in **`NULL`**, so we hear about everything (and hear about all IP addresses associated with the host name).
      - For the full list of legitimate service names, feel free to check out this link
    - The third argument is used for additional filter on the socket types supported by the host. By passing in **`NULL`** (as we will), we'll get information about all socket types. (More on sockets later on).
    - **`getaddrinfo`** implants the head of a linked list of **`struct addrinfo`** records in the space addresses by **`res`**.
  - **`freeaddrinfo`** fully disposes of the linked list surfaced by an earlier call to **`getaddrinfo`**.
  - Each node in the linked list includes an **`ai_addr`** field which can be passed in to **`getnameinfo`** to extract IP address information.
    - You can think of the **`ai_addr`** field (and the size of what **`ai_addr`** addresses, which is itself stored in **`ai_addrlen`**) as the payload that each node carries.
  - **`man`** pages exist for all three of these functions.

Resolving IP Addresses: Take II

- **Of course, there's the `struct addrinfo` type:**
  - The **`struct addrinfo`** and the hierarchy of relevant **`struct sockaddr`**-related records look like this:

```c
struct addrinfo {
  int ai_flags;             // input flags
  int ai_family;            // protocol family for socket
  int ai_socktype;          // socket type
  int ai_protocol;          // protocol for socket
  socklen_t ai_addrlen;     // length of socket-address
  struct sockaddr *ai_addr; // socket-address for socket
  char *ai_canonname;       // canonical name for service location
  struct addrinfo *ai_next; // pointer to next in list
};

struct sockaddr { // generic socket address record
  unsigned short sa_family; // protocol family for socket
  char sa_data[14];         // address data (and defines full size to be 16 bytes)
};

struct sockaddr_in { // IPv4 Internet-style socket address record
  unsigned short sin_family; // protocol family for socket
  unsigned short sin_port;   // port number (in network byte order)
  struct in_addr sin_addr;   // IP address (in network byte order)
  unsigned char sin_zero[8]; // pad to sizeof(struct sockaddr)
};

struct sockaddr_in6 { // IPv6 Internet-style socket address record
  unsigned short sin6_family;  // protocol family for socket
  unsigned short sin6_port;    // port number (in network byte order)
  // more fields, total size is > sizeof(struct sockaddr_in)
};
```

  - The **`ai_addr`** field within the **`addrinfo`** definition is statically typed to a **`struct sockaddr *`**, but in practice, the field stores the address of a record in the **`struct sockaddr`** *family*. The value in the first field—the **`unsigned short`**—is used to self-identify the exact record type being used. (And the **`ai_addrlen`** helps as well).
  - Not surprisingly, the **`ai_next`** field is used as a traditional next pointer that leads to the next **`addrinfo`** in the list.

Resolving IP Addresses: Take II

- **The three functions are the *official* way to get hostname information these days.**
  - In truth, I see more code written using the deprecated functions than I see using the more modern ones, because most know the old functions, and they're less concerned about building applications that are savvy to the needs to IPv6.
  - The textbook doesn't even mention the newer functions.
    - I'm only covering them because they're the official way.
    - You can using either set of functions and I'll be happy.
  - Here's the core of another program that uses these three functions to more or less do the same thing:

```
static void publishIPAddressInfo(const string& host) {
  struct addrinfo *infoList;
  int ret =
    getaddrinfo(host.c_str(), /* service = */ NULL,
                /* hints = */ NULL, &infoList);
  if (ret != 0) {
    cout << host << " could not be resolved to anything" << endl;
    return;
  }

  set<string> ipAddresses;
  addrinfo *curr = infoList;
  while (curr != NULL) {
    char ipAddress[64] = {'\0'};
    int ret =
      getnameinfo(curr->ai_addr, curr->ai_addrlen,
                  ipAddress, sizeof(ipAddress),
                  /* service = */ NULL, /* servlen = */ 0,
                  NI_NUMERICHOST);
    if (ret == 0 && ipAddress[0] != '\0' &&
        ipAddresses.find(ipAddress) == ipAddresses.cend()) {
      ipAddresses.insert(ipAddress);
      cout << "+ " << ipAddress << endl;
    }
    curr = curr->ai_next;
  }
  freeaddrinfo(infoList);
}
```

  - We rely on **`getnameinfo`** to populate the **`ipAddress`** buffer with the dotted-quad form of the IP address based on information present in the record addressed by **`infoList->ai_addr`**. We get the dotted- quad form of the host name instead of the human-readonable form, because the final argument—the value passed in through the **`flags`** paramater—was set to **`NI_NUMERICHOST`**.

Our first network client!

- **It's a network time client!**
  - At least during lecture, I'm running a laughably stupid time server on port 12345 of **`myth22.stanford.edu`**.
  - The client program illustrates the basics of creating a socket descriptor, establishing a connection to **`myth22.stanford.edu:12345`**, and then reading the very short response the server publishes to the socket the instant the time server detects a connection.
  - Here's the code that establishes a connection to the server/port pair of interest:

```
int createClientSocket(const string& host, unsigned short port) {
  struct hostent *he = gethostbyname(host.c_str());
  if (he == NULL) return kClientSocketError;

  int s = socket(AF_INET, SOCK_STREAM, 0);
  if (s < 0) return kClientSocketError;

  struct sockaddr_in serverAddress;
  memset(&serverAddress, 0, sizeof(serverAddress));
  serverAddress.sin_family = AF_INET;
  serverAddress.sin_port = htons(port);
  serverAddress.sin_addr.s_addr = ((struct in_addr *)he->h_addr)->s_addr;

  if (connect(s, (struct sockaddr *) &serverAddress,
              sizeof(serverAddress)) != 0) {
    close(s);
    return kClientSocketError;
  }

  return s;
}
```

  - At the end of it all, the **`s`** is the client-end socket descriptor that can be used to manage a two-way conversation with the service running on the remote host/port pair.

Our first network client! (continued)

- **Baby's First Protocol**
  - The full program file comes in three parts: here, here, and here
  - The protocol—informally, the set of rules both client and server must follow if they're to speak with one another—is simple.
  - The protocol here is...
    - The client connects (e.g. "rings" the service's phone at a particular "extension", and waits for the server to "pick up")
    - The client says nothing.
    - The server speaks by publishing the current time into its own end of the connection and then hangs up.
    - The client ingests the published text (understood, by protocol, to be just one line), publishes it to the console, and then itself hangs up.

```
int main(int argc, char *argv[]) {
  int clientSocket = createClientSocket("myth22.stanford.edu", 12345);
  if (clientSocket == kClientSocketError) {
    cerr << "Time server could not be reached" << endl;
    cerr << "Aborting" << endl;
    return kTimeServerInaccessible;
  }
  sockbuf sb(clientSocket);
  iosockstream ss(&sb);
  string timeline;
  getline(ss, timeline);
  cout << timeline << endl;
  return 0;
}
```

  - We'll first write the client using **`read`**, **`write`**, and **`close`** so we can speak to the complexities that come with using those functions.
    - We've written this type of code before, and written it in a way that seemed academic at the time.
    - It'll be the same code again, but this time it won't be academic, because the problems we need to protect against really could happen.
  - I'll then discuss the **`sockbuf`** and **`iosockstream`** classes, which come with an open source package called **`socket++`** which allows us to layer C++ streams over socket descriptors. #hugewin

Emulation of `wget`

- **`wget` is a command line utility that, given its URL, downloads a single document (HTML document, XML document, JPG, or whatever).**
  - Without being concerned so much about error checking and robustness, we can write something very simple to emulate the **`wget`**'s core functionality.
  - Full program is right here.
  - It'll allow me to illustrate the most basic parts of the HTTP protocol, which will become important once Assignment 6 is released next Wednesday.
  - I'll run **`/usr/bin/wget`** (the built-in) and **`web-get`** (which is what we'll write) side by side in lecture.
  - **`main`** entry point and parsing code is presented here:

```
static const string kProtocolPrefix = "http://";
static const string kDefaultPath = "/";
static pair<string, string> parseURL(string url) {
  if (startsWith(url, kProtocolPrefix)) // in "string-utils.h"
    url = url.substr(kProtocolPrefix.size());
  size_t found = url.find('/');
  if (found == string::npos)
    return make_pair(url, kDefaultPath); // defined in <utility>
  string host = url.substr(0, found);
  string path = url.substr(found);
  return make_pair(host, path);
}

int main(int argc, char *argv[]) {
  if (argc != 2) {
    cerr << "Usage: " << argv[0] << " <url>" << endl;
    return kWrongArgumentCount;
  }
  pullContent(parseURL(argv[1]));
  return 0;
}
```

  - The **`parseURL`** function is a gesture to the programmatic split at the host-path border. In practice, more protocols (**`https`**, for instance) might be supported as well.

Emulation of `wget` (continued)

- **Of course, the `pullContent` function needs to cover the networking.**
  - We've already written a **`createClientSocket`** function for our **`time-client`** program. Recognizing that function might be useful to a bunch of networked client applications, its interface and implementation were placed in standalone files.
  - There is, of course, **`web-get`**-specific work to be done:

```
static const unsigned short kDefaultHTTPPort = 80;
static void pullContent(const pair<string, string>& components) {
  int clientSocket = createClientSocket(components.first, kDefaultHTTPPort);
  // error checking omitted
  sockbuf sb(clientSocket);
  iosockstream ss(&sb);
  issueRequest(ss, components.first, components.second);
  skipHeader(ss);
  savePayload(ss, getFileName(components.second));
}
```

  - The implementations of **`issueRequest`**, **`skipHeader`**, and **`savePayload`** subdivide the client-server conversation into manageable chunks.
  - The best takeaway from the above is the fact that, for the second time in two examples, we've layered an **`iosockstream`** on top of a **`sockbuf`**, which itself is layered on top of our bidirectional socket descriptor.
    - Be glad we have the **`socket++`** library, because without it, we'd need to do so much manual character array manipulation that coding would cease to be fun.
    - One very relevant piece of information about the **`sockbuf`** class: its destructor closes the file descriptor passed to it when it's constructed, so we shouldn't call **`close`** on **`clientSocket`** ourselves.

Emulation of **`wget`** (continued)

- **The implementations of the helper functions are fairly straighforward:**
  - Here's the implementation of **`issueRequest`**. Notice that I manually construct the absolute minimum, two-line request imaginable and send it over the wire to the server.
  - It's standard HTTP-protocol practice that each line, including the blank line that marks the end of the request, end in CRLF (short for carriage-return-line-feed), which is '\r' following by '\n'.
  - The **`flush`** call is necessary to ensure all character data is pressed over the wire and consumable at the other end.

```
static void issueRequest(iosockstream& ss, const string& host, const string& path) {
  ss << "GET " << path << " HTTP/1.0\r\n";
  ss << "Host: " << host << "\r\n";
  ss << "\r\n";
  ss.flush();
}
```

  - After the **`flush`**, the client transitions from speak to listen mode.
    - The **`iosockstream`** is read/write, because the socket descriptor backing it is bidirectional.
  - We read in all of the HTTP response header lines until we get to either a blank line or one that contains nothing other than a '\r'.
  - The blank line is, indeed, supposed to be "\r\n", but some servers are sloppy, so we're supposed to treat the '\r' as optional. (Recall that **`getline`** chews up the '\n', but it'll leave '\r' at the end of the line).

```
static void skipHeader(iosockstream& ss) {
  string line;
  do {
    getline(ss, line);
  } while (!line.empty() && line != "\r");
}
```

  - This is a reasonably legitimate situation where the **`do/while`** loop is the correct idiom. #yaydowhileloops

Emulation of `wget` (continued)

- **Of course, there's that payload part.**
  - Everything beyond the response header and that blank line is considered payload—that's the file, the JSON, the HTML, the image, the cat video.
  - Every single byte that comes through should be replicated, in order, to a local copy.

```
static string getFileName(const string& path) {
  if (path.empty() || path[path.size() - 1] == '/') {
    return "index.html"; // not always correct, but not the point
  }

  size_t found = path.rfind('/');
  return path.substr(found + 1);
}

static const size_t kBufferSize = 1024; // just a random, large size
static void savePayload(iosockstream& ss, const string& filename) {
  ofstream output(filename, ios::binary); // don't assume it's text
  size_t totalBytes = 0;
  while (!ss.fail()) {
    char buffer[kBufferSize] = {'\0'};
    ss.read(buffer, sizeof(buffer));
    totalBytes += ss.gcount();
    output.write(buffer, ss.gcount());
  }
  cout << "Total number of bytes fetched: " << totalBytes << endl;
}
```

  - The HTTP/1.0 protocol dictates that everything beyond that blank line is payload, and that once the server publishes each and every byte of the payload, it closes it's end of the connection. That server-side close is the client-side's EOF, and we write everything we read.
    - Note that we open an **`ofstream`** in binary mode, predominantly so the **`ofstream`** doesn't do anything funky with byte characters that are *incidentally* newline characters.
    - **`gcount`** returns the number of bytes read by the most recent call to **`read`**. (This I did not know until I wrote this example).
  - The HTTP/1.1 protocol allows for connections to remain open, even after the initial payload has come through. I specifically avoid this here by going with HTTP/1.0, which doesn't allow this.